User's Manual

NetX™

User's Manual: Software

Renesas Synergy™ Platform

synergygallery.renesas.com

Renesas Electronics
www.renesas.com

Rev. 5

the high-performance real-time implementation
of TCP/IP standards

User Guide

Version 5

Express Logic, Inc.

858.613.6640
Toll Free 888.THREADX
FAX 858.521.4259
http://www.expresslogic.com

©2002-2010 by Express Logic, Inc.

All rights reserved. This document and the associated NetX software are the sole property of Express Logic, Inc. Each contains proprietary information of Express Logic, Inc. Reproduction or duplication by any means of any portion of this document without the prior written consent of Express Logic, Inc. is expressly forbidden.

Express Logic, Inc. reserves the right to make changes to the specifications described herein at any time and without notice in order to improve design or reliability of NetX. The information in this document has been carefully checked for accuracy; however, Express Logic, Inc. makes no warranty pertaining to the correctness of this document.

Trademarks

NetX, Piconet, and UDP Fast Path are trademarks of Express Logic, Inc. ThreadX is a registered trademark of Express Logic, Inc.

All other product and company names are trademarks or registered trademarks of their respective holders.

Warranty Limitations

Express Logic, Inc. makes no warranty of any kind that the NetX products will meet the USER's requirements, or will operate in the manner specified by the USER, or that the operation of the NetX products will operate uninterrupted or error free, or that any defects that may exist in the NetX products will be corrected after the warranty period. Express Logic, Inc. makes no warranties of any kind, either expressed or implied, including but not limited to the implied warranties of merchantability and fitness for a particular purpose, with respect to the NetX products. No oral or written information or advice given by Express Logic, Inc., its dealers, distributors, agents, or employees shall create any other warranty or in any way increase the scope of this warranty, and licensee may not rely on any such information or advice.

Part Number: 000-1009
Revision 5.3

Contents

About This Guide 9

- Guide Conventions 10
- NetX Data Types 11
- Customer Support Center 12

1 Introduction to NetX 15

- NetX Unique Features 16
- RFCs Supported by NetX 18
- Embedded Network Applications 19
- NetX Benefits 19

2 Installation and Use of NetX 23

- Host Considerations 24
- Target Considerations 24
- Product Distribution 25
- NetX Installation 26
- Using NetX 27
- Troubleshooting 27
- Configuration Options 28
- NetX Version ID 39

3 Functional Components of NetX 41

- Execution Overview 44
- Protocol Layering 48
- Packet Memory Pools 49
- Internet Protocol (IP) 58
- Address Resolution Protocol (ARP) 71
- Reverse Address Resolution Protocol (RARP) 75
- Internet Control Message Protocol (ICMP) 78
- Internet Group Management Protocol (IGMP) 81
- User Datagram Protocol (UDP) 85
- Transmission Control Protocol (TCP) 91

4 Description of NetX Services 105

5 NetX Network Drivers 343

- Driver Introduction 344
- Driver Entry 345
- Driver Requests 345
- Driver Output 358
- Driver Input 359
- Example RAM Ethernet Network Driver 361

A NetX Services 375

B NetX Constants 383

C NetX Data Types 403

D BSD-Compatible Socket API 411

E ASCII Character Codes 415

F Index 417

Figures

Figure 1 TCP/IP Protocol Layers 49

Figure 2 UDP Data Encapsulation 50

Figure 3 Packet Header and Memory Pool Layout 52

Figure 4 Network Packets and Chaining 56

Figure 5 IP Address Structure 59

Figure 6 IP Header Format 61

Figure 7 ARP Packet Format 76

Figure 8 ICMP Ping Message 79

Figure 9 IGMP Report Message 83

Figure 10 UDP Header 85

Figure 11 TCP Header 92

Figure 12 States of the TCP State Machine 96

NETX

About This Guide

This guide contains comprehensive information about NetX, the high-performance network stack from Express Logic, Inc.

It is intended for embedded real-time software developers familiar with basic networking concepts, the ThreadX RTOS, and the C programming language.

Organization

Chapter 1	Introduces NetX.
Chapter 2	Gives the basic steps to install and use NetX with your ThreadX application.
Chapter 3	Provides a functional overview of the NetX system and basic information about the TCP/IP networking standards.
Chapter 4	Details the application's interface to NetX.
Chapter 5	Describes network drivers for NetX.
Appendix A	NetX Services
Appendix B	NetX Constants
Appendix C	NetX Data Types
Appendix D	BSD-Compatible Socket API

Appendix D	ASCII Chart
Index	Topic cross reference

Guide Conventions

Italics	Typeface denotes book titles, emphasizes important words, and indicates variables.
Boldface	Typeface denotes file names, key words, and further emphasizes important words and variables.
	Information symbols draw attention to important or additional information that could affect performance or function.
	Warning symbols draw attention to situations that developers should avoid because they could cause fatal errors.

NetX Data Types

In addition to the custom NetX control structure data types, there are several special data types that are used in NetX service call interfaces. These special data types map directly to data types of the underlying C compiler. This is done to ensure portability between different C compilers. The exact implementation is inherited from ThreadX and can be found in the ***tx_port.h*** file included in the ThreadX distribution.

The following is a list of NetX service call data types and their associated meanings:

UINT	Basic unsigned integer. This type must support 8-bit unsigned data; however, it is mapped to the most convenient unsigned data type.
ULONG	Unsigned long type. This type must support 32-bit unsigned data.
VOID	Almost always equivalent to the compiler's void type.
CHAR	Most often a standard 8-bit character type.

Additional data types are used within the NetX source. They are located in either the ***tx_port.h*** or ***nx_port.h*** files.

Customer Support Center

Support engineers	858.613.6640
Support fax	858.521.4259
Support email	support@expresslogic.com
Web page	http://www.expresslogic.com

Latest Product Information

Visit the Express Logic web site and select the "Support" menu option to find the latest online support information, including information about the latest NetX product releases.

What We Need From You

To more efficiently resolve your support request, provide us with the following information in your email request:

1. A detailed description of the problem, including frequency of occurrence and whether it can be reliably reproduced.
2. A detailed description of any changes to the application and/or NetX that preceded the problem.
3. The contents of the ***_tx_version_id*** and ***_nx_version_id*** strings found in the ***tx_port.h*** and ***nx_port.h*** files of your distribution. These strings will provide us valuable information regarding your run-time environment.
4. The contents in RAM of the following ULONG variables:

   _tx_build_options
   _nx_system_build_options1
   _nx_system_build_options2

_nx_system_build_options3
_nx_system_build_options4
_nx_system_build_options5

These variables will give us information on how your ThreadX and NetX libraries were built.

5. A trace buffer captured immediately after the problem was detected. This is accomplished by building the ThreadX and NetX libraries with **TX_ENABLE_EVENT_TRACE** and calling ***tx_trace_enable*** with the trace buffer information. Refer to the *TraceX User Guide* for details.

Where to Send Comments About This Guide

The staff at Express Logic is always striving to provide you with better products. To help us achieve this goal, email any comments and suggestions to the Customer Support Center at

support@expresslogic.com

Please type “NetX User Guide” in the subject line.

CHAPTER 1

Introduction to NetX

NetX is a high-performance real-time implementation of the TCP/IP standards designed exclusively for embedded ThreadX-based applications. This chapter contains an introduction to NetX and a description of its applications and benefits.

- NetX Unique Features 16
  - Piconet™ Architecture 16
  - Zero-copy Implementation 16
  - UDP Fast Path™ Technology 17
  - ANSI C Source Code 17
  - Not A Black Box 17
  - BSD-Compatible Socket API 18
- RFCs Supported by NetX 18
- Embedded Network Applications 19
  - Real-time Network Software 19
- NetX Benefits 19
  - Improved Responsiveness 19
  - Software Maintenance 20
  - Increased Throughput 20
  - Processor Isolation 20
  - Ease of Use 20
  - Improve Time to Market 20
  - Protecting the Software Investment 21

NetX Unique Features

Unlike other TCP/IP implementations, NetX is designed to be versatile—easily scaling from small micro-controller-based applications to those that use powerful RISC and DSP processors. This is in sharp contrast to public domain or other commercial implementations originally intended for workstation environments but then squeezed into embedded designs.

Piconet™ Architecture

Underlying the superior scalability and performance of NetX is *Piconet*, a software architecture especially designed for embedded systems. Piconet architecture maximizes scalability by implementing NetX services as a C library. In this way, only those services actually used by the application are brought into the final runtime image. Hence, the actual size of NetX is completely determined by the application. For most applications, the instruction image requirements of NetX ranges between 5 KBytes and 30 KBytes in size.

NetX achieves superior network performance by layering internal component function calls only when it is absolutely necessary. In addition, much of NetX processing is done directly in-line, resulting in outstanding performance advantages over the workstation network software used in embedded designs in the past.

Zero-copy Implementation

NetX provides a packet-based, zero-copy implementation of TCP/IP. Zero copy means that data in the application's packet buffer are never copied inside NetX. This greatly improves performance and frees up valuable processor cycles

to the application, which is extremely important in embedded applications.

UDP Fast Path™ Technology

With *UDP Fast Path Technology*, NetX provides the fastest possible UDP processing. On the sending side, UDP processing—including the optional UDP checksum—is completely contained within the ***nx_udp_socket_send*** service. No additional function calls are made until the packet is ready to be sent via the internal NetX IP send routine. This routine is also flat (i.e., its function call nesting is minimal) so the packet is quickly dispatched to the application's network driver. When the UDP packet is received, the NetX packet-receive processing places the packet directly on the appropriate UDP socket's receive queue or gives it to the first thread suspended waiting for a receive packet from the UDP socket's receive queue. No additional ThreadX context switches are necessary. See page 17 for an example of Fast Path.

ANSI C Source Code

NetX is written completely in ANSI C and is portable immediately to virtually any processor architecture that has an ANSI C compiler and ThreadX support.

Not A Black Box

Most distributions of NetX include the complete C source code. This eliminates the "black-box" problems that occur with many commercial network stacks. By using NetX, applications developers can see exactly what the network stack is doing—there are no mysteries!

Having the source code also allows for application specific modifications. Although not recommended, it is certainly beneficial to have the ability to modify the network stack if it is required.

These features are especially comforting to developers accustomed to working with in-house or public domain network stacks. They expect to have source code and the ability to modify it. NetX is the ultimate network software for such developers.

BSD-Compatible Socket API

For legacy applications, NetX also provides a BSD-compatible socket interface that makes calls to the high-performance NetX API underneath. This helps in migrating existing network application code to NetX.

RFCs Supported by NetX

NetX support of RFCs describing basic network protocols includes but is not limited to the following network protocols. NetX follows all general recommendations and basic requirements within the constraints of a real-time operating system with small memory footprint and efficient execution.

RFC	Description	Page
RFC 1112	Host Extensions for IP Multicasting (IGMPv1)	81
RFC 2236	Internet Group Management Protocol, Version 2	81
RFC 768	User Datagram Protocol (UDP)	85
RFC 791	Internet Protocol (IP)	58
RFC 792	Internet Control Message Protocol(ICMP)	78
RFC 793	Transmission Control Protocol (TCP)	91
RFC 826	Ethernet Address Resolution Protocol(ARP)	71
RFC 903	Reverse Address Resolution Protocol (RARP)	75

Embedded Network Applications

Embedded network applications are applications that need network access and execute on microprocessors hidden inside products such as cellular phones, communication equipment, automotive engines, laser printers, medical devices, and so forth. Such applications almost always have some memory and performance constraints. Another distinction of embedded network applications is that their software and hardware have a dedicated purpose.

Real-time Network Software

Basically, network software that must perform its processing within an exact period of time is called *real-time network* software, and when time constraints are imposed on network applications, they are classified as real-time applications. Embedded network applications are almost always real-time because of their inherent interaction with the external world.

NetX Benefits

The primary benefits of using NetX for embedded applications are high-speed Internet connectivity and very small memory requirements. NetX is also completely integrated with the high-performance, multitasking ThreadX real-time operating system.

Improved Responsiveness

The high-performance NetX protocol stack enables embedded network applications to respond faster than ever before. This is especially important for embedded applications that either have a significant

volume of network traffic or stringent processing requirements on a single packet.

Software Maintenance

Using NetX allows developers to easily partition the network aspects of their embedded application. This partitioning makes the entire development process easy and significantly enhances future software maintenance.

Increased Throughput

NetX provides the highest-performance networking available, which directly transfers to the embedded application. NetX applications are able to process many more packets than non-NetX applications!

Processor Isolation

NetX provides a robust, processor-independent interface between the application and the underlying processor and network hardware. This allows developers to concentrate on the network aspects of the application rather than spending extra time dealing with hardware issues directly affecting networking.

Ease of Use

NetX is designed with the application developer in mind. The NetX architecture and service call interface are easy to understand. As a result, NetX developers can quickly use its advanced features.

Improve Time to Market

The powerful features of NetX accelerate the software development process. NetX abstracts most processor and network hardware issues, thereby removing these concerns from a majority of application network-specific areas. This, coupled with the ease-of-use and advanced feature set, result in a faster time to market!

Protecting the Software Investment

NetX is written exclusively in ANSI C and is fully integrated with the ThreadX real-time operating system. This means NetX applications are instantly portable to all ThreadX supported processors. Better still, a completely new processor architecture can be supported with ThreadX in a matter of weeks. As a result, using NetX ensures the application's migration path and protects the original development investment.

N E T X

CHAPTER 2

Installation and Use of NetX

This chapter contains a description of various issues related to installation, setup, and use of the high-performance network stack NetX, including the following:

- Host Considerations 24
- Target Considerations 24
- Product Distribution 25
- NetX Installation 26
- Using NetX 27
- Troubleshooting 27
- Configuration Options 28
  - System Configuration Options 29
  - ARP Configuration Options 30
  - ICMP Configuration Options 31
  - IGMP Configuration Options 32
  - IP Configuration Options 32
  - Packet Configuration Options 34
  - RARP Configuration Options 34
  - TCP Configuration Options 34
  - UDP Configuration Options 38
- NetX Version ID 39

Host Considerations

Embedded development is usually performed on Windows or Linux (Unix) host computers. After the application is compiled, linked, and located on the host, it is downloaded to the target hardware for execution.

Usually the target download is done from within the development tool's debugger. After download, the debugger is responsible for providing target execution control (go, halt, breakpoint, etc.) as well as access to memory and processor registers.

Most development tool debuggers communicate with the target hardware via on-chip debug (OCD) connections such as JTAG (IEEE 1149.1) and Background Debug Mode (BDM). Debuggers also communicate with target hardware through In-Circuit Emulation (ICE) connections. Both OCD and ICE connections provide robust solutions with minimal intrusion on the target resident software.

As for resources used on the host, the source code for NetX is delivered in ASCII format and requires approximately 1 Mbytes of space on the host computer’s hard disk.

*Please review the supplied **readme_netx.txt** file for additional host system considerations and options.*

Target Considerations

NetX requires between 5 KBytes and 30 KBytes of Read-Only Memory (ROM) on the target. Another 1 to 2 KBytes of the target’s Random Access Memory

(RAM) are required for the NetX thread stack and other global data structures.

In addition, NetX requires the use of a ThreadX timer and a ThreadX mutex object. These facilities are used for periodic processing needs and thread protection inside the NetX protocol stack.

Product Distribution

Two types of NetX packages are available—*standard* and *premium*. The *standard* package includes minimal source code, while the *premium* package contains complete NetX source code. Either package is shipped on a single CD.

The exact contents of the distribution CD depends on the target processor, development tools, and the NetX package purchased. Following is a list of the important files common to most product distributions:

readme_netx.txt
: This file contains specific information about the NetX port, including information about the target processor and the development tools.

nx_api.h
: This C header file contains all system equates, data structures, and service prototypes.

nx_port.h
: This C header file contains all development tool specific data definitions and structures.

demo_netx.c
: This C file contains a small demo application.

nx.a (or nx.lib) This is the binary version of the NetX C library. It is distributed with the *standard* package.

All files are in lower-case, making it easy to convert the commands to Linux (Unix) development platforms.

NetX Installation

Installation of NetX is straightforward. The following instructions apply to virtually any installation. However, examine the ***readme_netx.txt*** file for changes specific to the actual development tool environment.

Backup the NetX distribution disk and store it in a safe location.

On the host hard drive, copy all the files of the NetX distribution into the previously created and installed ThreadX directory.

If installing the *standard* package, NetX installation is now complete. Otherwise, if installing the premium package, you must build the NetX runtime library.

*Application software needs access to the NetX library file, usually called **nx.a** (or **nx.lib**), and the C include files **nx_api.h** and **nx_port.h**. This is accomplished either by setting the appropriate path for the development tools or by copying these files into the application development area.*

Using NetX

Using NetX is easy. Basically, the application code must include ***nx_api.h*** during compilation and link with the NetX library ***nx.a*** (or ***nx.lib***).

There are four easy steps required to build a NetX application:

Step 1: Include the ***nx_api.h*** file in all application files that use NetX services or data structures.

Step 2: Initialize the NetX system by calling ***nx_system_initialize*** from the ***tx_application_define*** function or an application thread.

Step 3: Create an IP instance, enable the Address Resolution Protocol (ARP), if necessary, and any sockets after ***nx_system_initialize*** is called.

Step 4: Compile application source and link with the NetX runtime library ***nx.a*** (or ***nx.lib***). The resulting image can be downloaded to the target and executed!

Troubleshooting

Each NetX port is delivered with a demonstration application that executes with a simulated network driver. This same demonstration is delivered with all versions of NetX and provides the ability to run NetX without any network hardware. It is always a good idea to get the demonstration system running first.

*See the **readme_netx.txt** file supplied with the distribution for more specific details regarding the demonstration system.*

If the demonstration system does not run properly, perform the following operations to narrow the problem:

1. Determine how much of the demonstration is running.
2. Increase stack sizes in any new application threads.
3. Recompile the NetX library with the appropriate debug options listed in the configuration option section.
4. Examine the NX_IP structure to see if packets are being sent or received.
5. Examine the default packet pool to see if there are available packets.
6. Ensure network driver is supplying ARP and IP packets with their headers on 4-byte boundaries.
7. Temporarily bypass any recent changes to see if the problem disappears or changes. Such information should prove useful to Express Logic support engineers.

Follow the procedures outlined in the “What We Need From You” on page 12 to send the information gathered from the troubleshooting steps.

Configuration Options

There are several configuration options when building the NetX library and the application using NetX. The options below can be defined in the application source, on the command line, or within the ***nx_user.h*** include file.

Options defined in ***nx_user.h*** are applied only if the application and NetX library are built with ***NX_INCLUDE_USER_DEFINE_FILE*** defined.

Review the ***readme_netx.txt*** file for additional options for your specific version of NetX. The following sections describe the configuration options available in NetX:

System Configuration Options

Define	Meaning
NX_DEBUG	Defined, this option enables the optional print debug information available from the RAM Ethernet network driver.
NX_DEBUG_PACKET	Defined, this option enables the optional debug packet dumping available in the RAM Ethernet network driver.
NX_DISABLE_ERROR_CHECKING	Defined, this option removes the basic NetX error checking API and results in a 15-percent performance improvement. API return codes not affected by disabling error checking are listed in bold typeface in the API definition. This define is typically used after the application is debugged sufficiently and its use improves performance and decreases code size.
NX_DRIVER_DEFERRED_PROCESSING	Defined, this option enables deferred network driver packet handling. This allows the network driver to place a packet on the IP instance and have the network's real processing routine called from the NetX internal IP helper thread.

Define	Meaning
NX_LITTLE_ENDIAN	Defined, this option performs the necessary byte swapping on little endian environments to ensure the protocol headers are in proper big endian format. Note that the default is typically setup in ***nx_port.h***.
NX_MAX_PHYSICAL_INTERFACES	Specifies the total number of physical network interfaces on the host device. The default value is 1; a host must have at least one physical interface. Note this does not include the loopback interface.
NX_PHYSICAL_HEADER	Specifies the size in bytes of the physical packet header. The default value is 16 (based on a typical 16-byte Ethernet frame) and is defined in ***nx_api.h***. The application can override the default by defining the value before ***nx_api.h*** is included.
NX_PHYSICAL_TRAILER	Specifies the size in bytes of the physical packet trailer and is typically used to reserve storage for things like Ethernet CRCs, etc. The default value is 4.

ARP Configuration Options

Define	Meaning
NX_DISABLE_ARP_INFO	Defined, this option disables ARP information gathering.
NX_ARP_DISABLE_AUTO_ARP_ENTRY	Defined, this option disables entering ARP request information in the ARP cache.

NX_ARP_EXPIRATION_RATE	This define specifies the number of seconds ARP entries remain valid. The default value of zero disables expiration or aging of ARP entries and is defined in ***nx_api.h***. The application can override the default by defining the value before ***nx_api.h*** is included.
NX_ARP_MAX_QUEUE_DEPTH	This defines specifies the maximum number of packets that can be queued while waiting for an ARP response. The default value is 4.
NX_ARP_MAXIMUM_RETRIES	This define specifies the maximum number of ARP retries made without an ARP response. The default value is 18 and is defined in ***nx_api.h***. The application can override the default by defining the value before ***nx_api.h*** is included.
NX_ARP_UPDATE_RATE	This define specifies the number of seconds between ARP retries. The default value is 10, which represents 10 seconds, and is defined in ***nx_api.h***. The application can override the default by defining the value before ***nx_api.h*** is included.

ICMP Configuration Options

Define	Meaning
NX_DISABLE_ICMP_INFO	Defined, this option disables ICMP information gathering.
NX_ICMP_ENABLE_DEBUG_LOG	Defined, this option enables the optional ICMP debug log.

IGMP Configuration Options

Define	Meaning
NX_DISABLE_IGMP_INFO	Defined, this option disables IGMP information gathering.
NX_DISABLE_IGMPV2	Defined, IGMP v2 support is disabled.
NX_IGMP_ENABLE_DEBUG_LOG	Defined, this option enables the optional IGMP debug log.
NX_MAX_MULTICAST_GROUPS	This define specifies the maximum number of multicast groups that can be joined. The default value is 7 and is defined in ***nx_api.h***. The application can override the default by defining the value before ***nx_api.h*** is included.

IP Configuration Options

Define	Meaning
NX_DISABLE_FRAGMENTATION	This define disables IP fragmentation logic.
NX_DISABLE_IP_INFO	Defined, this option disables IP information gathering.
NX_DISABLE_IP_RX_CHECKSUM	Defined, this option disables checksum logic on received IP packets. This is useful if the link-layer has reliable checksum or CRC logic.
NX_DISABLE_IP_TX_CHECKSUM	Defined, this option disables checksum logic on IP packets sent. This is only useful in situations in which the receiving network node has received IP checksum logic disabled.

NX_DISABLE_LOOPBACK_INTERFACE	Defined, this option disables NetX support on the 127.0.0.1 loopback interface. The 127.0.0.1 loopback interface is enabled by default.
NX_DISABLE_RX_SIZE_CHECKING	Defined, this option disables the addition size checking on received packets.
NX_ENABLE_IP_STATIC_ROUTING	Defined, this enables static routing in which a destination address can be assigned a specific next hop address. The default is that static routing is disabled.
NX_IP_ENABLE_DEBUG_LOG	Defined, this option enables the optional IP debug log.
NX_IP_PERIODIC_RATE	This define specifies the number of ThreadX timer ticks in one second. The default value is 100 (based on a 10ms ThreadX timer interrupt) and is defined in ***nx_port.h***. The application can override the default by defining the value before ***nx_api.h*** is included.
NX_IP_ROUTING_TABLE_SIZE	This defines the maximum number of entries in the routing table, which is a list of an outgoing interface and the next hop addresses for a given destination address. The default value is 8.
NX_MAX_IP_INTERFACES	The total number of logical network interfaces on the host device. The default value depends if the loopback interface is enabled. If so, the default value is 2, a physical interface and the loopback interface. Otherwise the default value is 1 for the sole physical interface.

Packet Configuration Options

Define	Meaning
NX_DISABLE_PACKET_INFO	Defined, this option disables packet pool information gathering.
NX_PACKET_ENABLE_DEBUG_LOG	Defined, this option enables the optional packet debug log.

RARP Configuration Options

Define	Meaning
NX_DISABLE_RARP_INFO	Defined, this option disables RARP information gathering.
NX_RARP_ENABLE_DEBUG_LOG	Defined, this option enables the optional RARP debug log.

TCP Configuration Options

Define	Meaning
NX_DISABLE_RESET_DISCONNECT	Defined, this option disables the reset processing during disconnect when the timeout value supplied is specified as NX_NO_WAIT.
NX_DISABLE_TCP_INFO	Defined, this option disables TCP information gathering.

NX_DISABLE_TCP_RX_CHECKSUM	Defined, this option disables checksum logic on received TCP packets. This is only useful in situations in which the link-layer has reliable checksum or CRC processing.
NX_DISABLE_TCP_TX_CHECKSUM	Defined, this option disables checksum logic for sending TCP packets. This is only useful in situations in which the receiving network node has received TCP checksum logic disabled.
NX_MAX_LISTEN_REQUESTS	This define specifies the maximum number of server listen requests. The default value is 10 and is defined in ***nx_api.h***. The application can override the default by defining the value before ***nx_api.h*** is included.
NX_TCP_ACK_EVERY_N_PACKETS	This specifies the number of TCP packets to receive before sending an ACK. The default value is 2 where an ACK packet is sent for every 2 packets received. Note if NX_TCP_IMMEDIATE_ACK is enabled but NX_TCP_ACK_EVERY_N_PACKETS is not, this value is automatically set to 1 for backward compatibility.
NX_TCP_ACK_TIMER_RATE	This define specifies how the number of system ticks (NX_IP_PERIODIC_RATE) is divided to calculate the timer rate for the TCP delayed ACK processing. The default value is 5, which represents 200ms, and is defined in ***nx_tcp.h***. The application can override the default by defining the value before ***nx_api.h*** is included.
NX_TCP_ENABLE_DEBUG_LOG	Defined, this option enables the optional TCP debug log.
NX_TCP_ENABLE_KEEPALIVE	Defined, this option enables the optional TCP keepalive timer.

NX_TCP_FAST_TIMER_RATE	This define specifies how the number of system ticks (NX_IP_PERIODIC_RATE) is divided to calculate the fast TCP timer rate. The fast TCP timer is used to drive the various TCP timers, including the delayed ACK timer. The default value is 10, which represents 100ms, and is defined in ***nx_tcp.h***. The application can override the default by defining the value before ***nx_api.h*** is included.
NX_TCP_IMMEDIATE_ACK	Defined, this option enables the optional TCP immediate ACK response processing.
NX_TCP_KEEPALIVE_INITIAL	This define specifies how many seconds of inactivity before the keepalive timer activates. The default value is 7200, which represents 2 hours, and is defined in ***nx_tcp.h***. The application can override the default by defining the value before ***nx_api.h*** is included.
NX_TCP_KEEPALIVE_RETRY	This define specifies how many seconds between retries of the keepalive timer assuming the other side of the connection is not responding. The default value is 75, which represents 75 seconds between retries, and is defined in ***nx_tcp.h***. The application can override the default by defining the value before ***nx_api.h*** is included.
NX_TCP_KEEPALIVE_RETRIES	This define specifies how many keepalive retries are allowed before the connection is deemed broken. The default value is 10, which represents 10 retries, and is defined in ***nx_tcp.h***. The application can override the default by defining the value before ***nx_api.h*** is included.

NX_TCP_MAXIMUM_RETRIES	This define specifies how many transmit retries are allowed before the connection is deemed broken. The default value is 10, which represents 10 retries, and is defined in ***nx_tcp.h***. The application can override the default by defining the value before ***nx_api.h*** is included.
NX_TCP_MAXIMUM_TX_QUEUE	This define specifies the maximum depth of the TCP transmit queue before TCP send requests are suspended or rejected. The default value is 20, which means that a maximum of 20 packets can be in the transmit queue at any given time. Note that packets stay in the transmit queue until an ACK is received from the other side of the connection. This constant is defined in ***nx_tcp.h***. The application can override the default by defining the value before ***nx_api.h*** is included.

NX_TCP_RETRY_SHIFT	This define specifies how the retransmit timeout period changes between retries. If this value is 0, the initial retransmit timeout is the same as subsequent retransmit timeouts. If this value is 1, each successive retransmit is twice as long. If this value is 2, each subsequent retransmit timeout is four times as long. The default value is 0 and is defined in ***nx_tcp.h***. The application can override the default by defining the value before ***nx_api.h*** is included.
NX_TCP_TRANSMIT_TIMER_RATE	This define specifies how the number of system ticks (NX_IP_PERIODIC_RATE) is divided to calculate the timer rate for the TCP transmit retry processing. The default value is 1, which represents 1 second, and is defined in ***nx_tcp.h***. The application can override the default by defining the value before ***nx_api.h*** is included.

UDP Configuration Options

Define	Meaning
NX_DISABLE_UDP_INFO	Defined, this option disables UDP information gathering.
NX_UDP_ENABLE_DEBUG_LOG	Defined, this option enables the optional UDP debug log.

*Additional development tool options are described in the **readme_netx.txt** file supplied on the distribution disk.*

NetX Version ID

The current version of NetX is available to both the user and the application software during runtime. The programmer can find the NetX version in the ***readme_netx.txt*** file. This file also contains a version history of the corresponding port. Application software can obtain the NetX version by examining the global string ***_nx_version_id***.

CHAPTER 3

Functional Components of NetX

This chapter contains a description of the high-performance NetX TCP/IP stack from a functional perspective.

- Execution Overview 44
  - Initialization 44
  - Application Interface Calls 45
  - Internal IP Thread 45
  - IP Periodic Timers 47
  - Network Driver 47
- Protocol Layering 48
- Packet Memory Pools 49
  - Creating Packet Pools 51
  - Packet Header NX_PACKET 51
  - Pool Capacity 55
  - Packet Pool Memory Area 57
  - Thread Suspension 57
  - Pool Statistics and Errors 57
  - Packet Pool Control Block NX_PACKET_POOL 58
- Internet Protocol (IP) 58
  - IP Addresses 58
  - Gateway IP Address 60
  - IP Header 60
  - IP Fragmentation 63
  - IP Send 64
  - IP Receive 65
  - Raw IP Send 65
  - Raw IP Receive 66
  - Creating IP Instances 66
  - Default Packet Pool 67
  - IP Helper Thread 67
  - Thread Suspension 67
  - IP Statistics and Errors 68
  - IP Control Block NX_IP 68

Multiple Network Interface (Multihome) Support 69
Static IP Routing 70

- Address Resolution Protocol (ARP) 71
  - ARP Enable 72
  - ARP Cache 72
  - ARP Dynamic Entries 72
  - ARP Static Entries 72
  - ARP Messages 73
  - ARP Aging 74
  - ARP Statistics and Errors 75
- Reverse Address Resolution Protocol (RARP) 75
  - RARP Enable 76
  - RARP Request 76
  - RARP Reply 77
  - RARP Statistics and Errors 78
- Internet Control Message Protocol (ICMP) 78
  - ICMP Enable 78
  - Ping Request 78
  - Ping Response 80
  - Thread Suspension 80
  - ICMP Statistics and Errors 80
- Internet Group Management Protocol (IGMP) 81
  - IGMP Enable 81
  - Multicast IP Addresses 81
  - Physical Address Mapping 82
  - Multicast Group Join 82
  - Multicast Group Leave 82
  - IGMP Report Message 83
  - IGMP Statistics and Errors 84
- User Datagram Protocol (UDP) 85
  - UDP Enable 85
  - UDP Header 85
  - UDP Checksum 86
  - UDP Ports and Binding 87
  - UDP Fast Path™ 87
  - UDP Packet Send 87
  - UDP Packet Receive 88
  - UDP Receive Notify 89
  - UDP Socket Create 89

Thread Suspension 89
UDP Socket Statistics and Errors 89
UDP Socket Control Block TX_UDP_SOCKET 90
Transmission Control Protocol (TCP) 91
TCP Enable 91
TCP Header 91
TCP Checksum 93
TCP Ports 94
Client Server Model 94
TCP Socket State Machine 94
TCP Client Connection 95
TCP Client Disconnection 95
TCP Server Connection 97
TCP Server Disconnection 99
Stop Listening on a Server Port 100
TCP Window Size 100
TCP Packet Send 100
TCP Packet Retransmit 101
TCP Packet Receive 101
TCP Receive Notify 101
TCP Socket Create 102
Thread Suspension 102
TCP Socket Statistics and Errors 102
TCP Socket Control Block NX_TCP_SOCKET 103

Execution Overview

There are five types of program execution within a NetX application: initialization, application interface calls, internal IP thread, IP periodic timers, and the network driver.

NetX assumes the existence of ThreadX and depends on its thread execution, suspension, periodic timers, and mutual exclusion facilities.

Initialization

The service ***nx_system_initialize*** must be called before any other NetX service is called. System initialization can be called either from the ThreadX ***tx_application_define*** routine or from application threads.

After ***nx_system_initialize*** returns, the system is ready to create packet pools and IP instances. Because creating an IP instance requires a default packet pool, at least one NetX packet pool must exist prior to creating an IP instance. Creating packet pools and IP instances is allowed from the ThreadX initialization function ***tx_application_define*** and from application threads.

Internally, creating an IP instance is accomplished in two parts: The first part is done within the context of the caller, either from ***tx_application_define*** or from an application thread's context. This includes setting up the IP data structure and creating various IP resources, including the internal IP thread. The second part is performed during the initial execution from the internal IP thread. This is where the application's network driver, supplied during the first part of IP creation, is first called. Calling the network driver from the internal IP thread enables the network driver to perform I/O and suspend during its

initialization processing. When the network driver returns from its initialization processing, the IP creation is complete.

*The NetX service **nx_ip_status_check** is available to obtain information on the IP instance (primary interface) status such as if the link is initialized, enabled and IP address is resolved. This information is used to synchronize application threads needing to use a newly created IP instance. For multihome hosts, **nx_ip_interface_status_check** is available to obtain information on the specified interface status..*

Application Interface Calls

Calls from the application are largely made from application threads running under the ThreadX RTOS. However, some initialization, create, and enable services may be called from ***tx_application_define***. The “Allowed From” sections in Chapter 4 (page 89) indicate from which each NetX service can be called from.

For the most part, processing intensive activities such as computing checksums is done within the calling thread’s context—without blocking access of other threads to the IP instance. For example, UDP checksum calculation is performed inside the ***nx_udp_socket_send*** service, prior to calling the underlying IP send function. On a received packet, the UDP checksum is calculated in the ***nx_udp_socket_receive*** service. This helps prevent stalling network requests of higher-priority threads because of processing intensive checksum processing in lower-priority threads.

Internal IP Thread

As mentioned, each IP instance in NetX has its own thread. The priority and stack size of the internal IP thread is defined in the ***nx_ip_create*** service. The internal IP thread is created in a ready-to-execute

mode. If the IP thread has a higher priority than the calling thread, preemption may occur inside the IP create call.

The entry point of the internal IP thread is at the function ***_nx_ip_thread_entry***. When started, the internal IP thread first completes network driver initialization, which consists of making two calls to the application-specific network driver. The first call is made to initialize the network driver. After the network driver returns from initialization (it may suspend while waiting for the hardware to be properly set up), the internal IP thread calls the network driver again to enable the link. After the network driver returns from the link enable call, the internal IP thread enters a while-forever loop checking for various events that need processing for this IP instance. Events processed in this loop include deferred IP packet reception, ARP packet processing, IP packet fragment assembly, ICMP ping processing, IGMP processing, TCP packet queue processing, TCP periodic processing, ARP periodic processing, IP fragment assembly timeouts, and IGMP periodic processing.

For multihome hosts, ***_nx_ip_thread_entry*** loops through each physical interface attached to the IP instance to initialize and enable the driver. The internal IP thread checks for events on each interface while processing in its while-forever loop. For certain events, IP thread performs the same action for each interface. These include deferred processing requests and IGMP enable events.

The NetX callback functions, including listen and disconnect callbacks, are called from the internal IP thread—not the original calling thread. The application must take care not to suspend inside any NetX callback function.

IP Periodic Timers

There is one ThreadX periodic timer used for each IP instance. This first one-second periodic timer performs ARP, IGMP, TCP timeout, and IP fragment timeout processing.

Network Driver

Each IP instance in NetX has a primary interface network driver specified by the application in the ***nx_ip_create*** service. The network driver is responsible for handling various NetX requests, including packet transmission, packet reception, and various requests for status and control. On transmission, the network driver is also responsible for buffering packets that cannot be immediately sent through the physical hardware.

For multihome hosts, each additional interface associated with the IP instance has an associated network driver that performs these tasks for the respective interface. Some drivers are written to handle two or more physical interfaces.

Single interface host applications need not make any changes to their existing drivers.

The network driver must also handle asynchronous events occurring on the media. Asynchronous events from the media include packet reception, packet transmission completion, and status changes. NetX provides the network driver with several access functions to handle various received packets. These functions are designed to be called from the interrupt service routine portion of the network driver. The network driver should forward all ARP packets received to the ***_nx_arp_packet_deferred_receive*** function. All RARP packets should be forwarded to ***_nx_rarp_packet_deferred_receive***. There are two options for IP packets. If fast dispatch of IP packets is required, incoming IP packets should be forwarded to ***_nx_ip_packet_receive*** for immediate processing.

This greatly improves NetX performance in handling IP packets and UDP packets. Otherwise, forwarding IP packets to ***_nx_ip_packet_deferred_receive*** should be done. This service places the IP packet in the deferred processing queue where it is then handled by the internal IP thread, which results in the least amount of ISR processing time.

The network driver can also defer interrupt processing to run out of the context of the IP thread. This is accomplished by calling the ***_nx_ip_driver_deferred_processing*** function from the network driver's interrupt routine.

See Chapter 5, “NetX Network Drivers” on page 343 for more detailed information on writing NetX network drivers.

Protocol Layering

The TCP/IP implemented by NetX is a layered protocol, which means more complex protocols are built on top of simpler underlying protocols. In TCP/IP, the lowest layer protocol is at the *link level* and is handled by the network driver. This level is typically targeted towards Ethernet, but it could also be fiber, serial, or virtually any physical media.

On top of the link layer is the *network layer*. In TCP/IP, this is the IP, which is basically responsible for sending and receiving simple packets—in a best-effort manner—across the network. Management-type protocols like ICMP and IGMP are typically also categorized as network layers, even though they rely on IP for sending and receiving.

The *transport layer* rests on top of the network layer. This layer is responsible for managing the flow of

data between hosts on the network. There are two types of transport services in TCP/IP: UDP and TCP. UDP services provide best-effort sending and receiving of data between two hosts in a connectionless manner, while TCP provides connection management between two host entities with a reliable data path between them.

This layering is reflected in the actual network data packets. Each layer in TCP/IP contains a block of information called a header. This technique of surrounding data (and possibly protocol information) with a header is typically called data encapsulation. Figure 1 shows an example of NetX layering and Figure 2 shows the resulting data encapsulation for UDP data being sent.

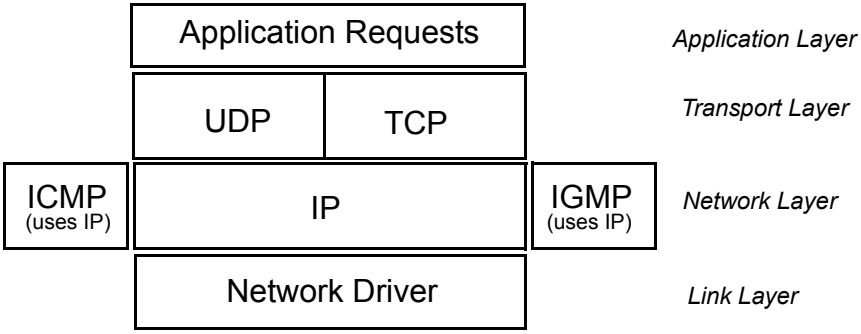

FIGURE 1. TCP/IP Protocol Layers

Packet Memory Pools

Allocating memory packets in a fast and deterministic manner is always a challenge in real-time networking applications. With this in mind, NetX provides the

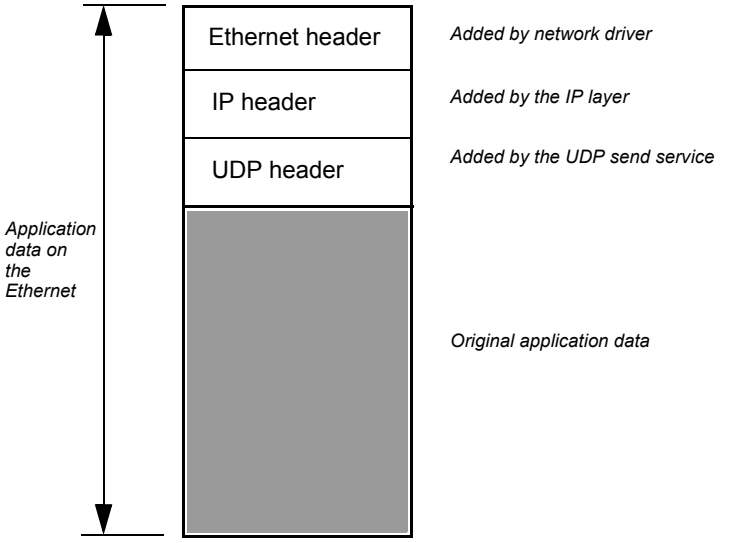

FIGURE 2. UDP Data Encapsulation

ability to create and manage multiple pools of fixed-size network packets.

Because NetX packet pools consist of fixed-size memory blocks, there are never any fragmentation problems. Of course, fragmentation causes behavior that is inherently indeterministic. In addition, the time required to allocate and free a NetX packet amounts to simple linked-list manipulation. Furthermore, packet allocation and deallocation is done at the head of the available list. This provides the fastest possible linked list processing.

Lack of flexibility is typically the main drawback of fixed-size packet pools. Determining the optimal packet payload size that also handles the worst-case incoming packet is a difficult task. NetX packets address this problem with packet chaining. An actual network packet can be made of one or more NetX

packets linked together. In addition, the packet header maintains a pointer to the top of the packet. As additional protocols are added, this pointer is simply moved backwards and the new header is written directly in front of the data. Without the flexible packet technology, the stack would have to allocate another buffer and copy the data into a new buffer with the new header, which is processing intensive.

Each NetX packet memory pool is a public resource. NetX places no constraints on how packet pools are used.

Creating Packet Pools

Packet memory pools are created either during initialization or during runtime by application threads. There are no limits on the number of packet memory pools in a NetX application.

Packet Header NX_PACKET

By default, NetX places the packet header immediately before the packet payload area. The packet memory pool is basically a series of packets—headers followed immediately by the packet payload. The packet header (***NX_PACKET***)

and the layout of the packet pool are pictured in Figure 3.

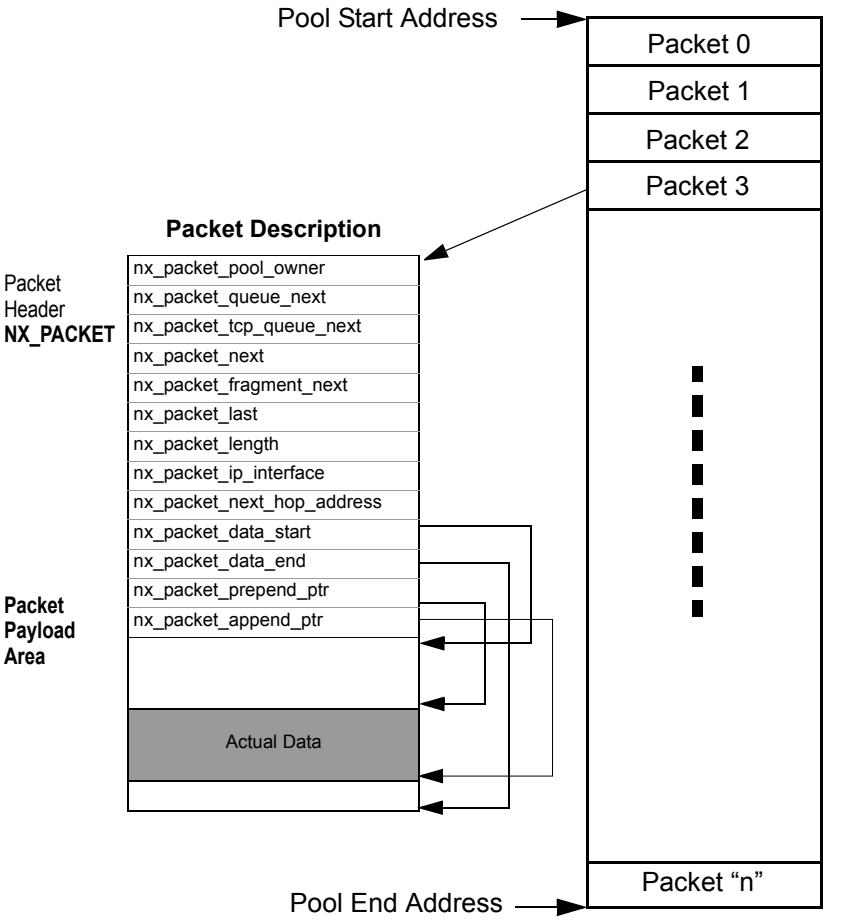

FIGURE 3. Packet Header and Memory Pool Layout

The fields of the packet header are defined as follows:

*It is important for the network driver to use the **nx_packet_transmit_release** function when transmission of a packet is complete. This function checks to make sure the packet is not part of a TCP output queue before it is actually placed back in the available pool.*

Packet header	Purpose
nx_packet_pool_owner	This field points to the owner of this particular packet. When the packet is released, it is released to this particular pool. With the pool ownership inside each packet, it is possible for a datagram to span multiple packets from multiple packet pools.
nx_packet_queue_next	This field points to the first packet of the next separate network packet. If NULL, there is no next network packet. This field is used by NetX to queue network packets, and it is also available to the network driver to queue packets for transmission.
nx_packet_tcp_queue_next	This field points to the first packet of the next separate TCP network packet on a specific socket's output queue. This requires a separate pointer because TCP packets are retransmitted if an ACK is not received from the connection prior to a specific timeout. If this field contains the constant NX_PACKET_FREE or NX_PACKET_ALLOCATED, then the network packet is not part of a TCP queue.
nx_packet_next	This field points to the next packet within the same network packet. If NULL, there are no additional packets that are part of the network packet. This field is also used to hold fragmented packets until the entire packet can be re-assembled.
nx_packet_last	This field points to the last packet within the same network packet. If NULL, this packet represents the entire network packet.

Packet header	Purpose
nx_packet_fragment_next	This field is used to hold different incoming IP packets in the process of being unfragmented.
nx_packet_length	This field contains the total number of bytes in the entire network packet, including the total of all bytes in all packets chained together by the *nx_packet_next* member.
nx_ip_interface	This field is the interface control block which is assigned to the packet when it is received by the interface driver, and by NetX for outgoing packets.
nx_next_hop_address	This field is used by the transmission logic. The next hop address determines how NetX forwards the packet to the final destination. If the destination address is on the local network, the next hop address is the same as the destination address. Otherwise the next hop address would be the router that knows how to forward the packet to the destination.
nx_packet_data_start	This pointer field points to the start of the physical payload area of this packet. It does not have to be immediately following the NX_PACKET header, but that is the default for the *nx_packet_pool_create* service.
nx_packet_data_end	This pointer field points to the end of the physical payload area of this packet. The difference between this field and the *nx_packet_data_start* field represents the payload size.

Packet header	Purpose
nx_packet_prepend_ptr	This pointer field points to the location of where packet data—either protocol header or actual data—is added in front of the existing packet data (if any) in the packet payload area. It must be greater than the *nx_packet_data_start* pointer location and less than or equal to the *nx_packet_append_ptr* pointer. *For performance reasons, NetX assumes* ***nx_packet_prepend_ptr*** *always points to a long-word boundary. Hence, any manipulation of this field must maintain this long-word alignment.*
nx_packet_append_ptr	This pointer field points to the end of the data currently in the packet payload area. It must be less than or equal to the *nx_packet_data_end* pointer. The difference between this field and the *nx_packet_data_start* field represents the amount of data in this packet.

Figure 4 shows three network packets in a queue, in which the middle network packet is composed of two packet structures. This illustrates how NetX achieves zero-copy performance by chaining together fixed-size packet structures. It also shows how NetX packet queues are independent from individual packet chains.

Pool Capacity

The number of packets in a packet pool is a function of the payload size and the total number of bytes in the memory area supplied to the packet pool create service. The capacity of the pool is calculated by dividing the packet size (including the size of the NX_PACKET header, the payload size, and any necessary padding to keep long-word alignment) into the total number of bytes in the supplied memory area.

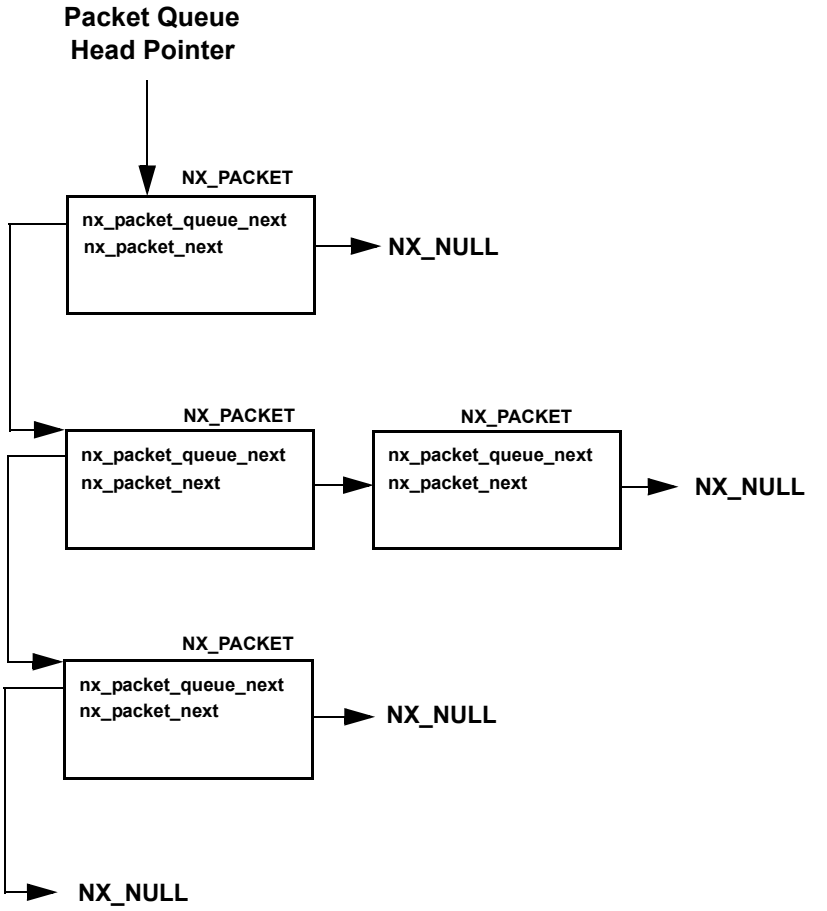

FIGURE 4. Network Packets and Chaining

Although NetX packet pool create function allocates the payload area immediately following the packet header, it is possible for the application to create packet pools where the payload is in a separate memory area from the packet headers. The only complication with this technique is calculating the header pointer again given just the starting address of the payload. This situation typically occurs inside the receive packet interrupt processing of the

network driver. If the packet payload immediately follows the header, the packet header is easily calculated by just moving backwards the size of the packet header. However, if the payload is in a different memory space from its header, the header would need to be calculated by examination of the relative offset of the payload and then applying that same offset to the start of the pool's packet header area.

Packet Pool Memory Area

The memory area for the packet pool is specified during creation. Like other memory areas for ThreadX and NetX objects, it can be located anywhere in the target's address space.

This is an important feature because of the considerable flexibility it gives the application. For example, suppose that a communication product has a high-speed memory area for network buffers. This memory area is easily managed by making it into a NetX packet memory pool.

Thread Suspension

Application threads can suspend while waiting for a packet from an empty pool. When a packet is returned to the pool, the suspended thread is given this packet and resumed.

If multiple threads are suspended on the same packet pool, they are resumed in the order they were suspended (FIFO).

Pool Statistics and Errors

If enabled, the NetX packet management software keeps track of several statistics and errors that may be useful to the application. The following statistics and error reports are maintained for packet pools:

Total Packets in Pool
Free Packets in Pool

Total Packet Allocations
Pool Empty Allocation Requests
Pool Empty Allocation Suspensions
Invalid Packet Releases

All of these statistics and error reports are available to the application with the ***nx_packet_pool_info_get*** service.

Packet Pool Control Block NX_PACKET_POOL

The characteristics of each packet memory pool are found in its control block. It contains useful information such as the linked list of free packets, the number of free packets, and the payload size for packets in this pool. This structure is defined in the ***nx_api.h*** file.

Packet pool control blocks can be located anywhere in memory, but it is most common to make the control block a global structure by defining it outside the scope of any function.

Internet Protocol (IP)

The Internet Protocol (IP) component of NetX is responsible for sending and receiving packets on the Internet (RFC 791). In NetX, it is the component ultimately responsible for sending and receiving TCP, UDP, ICMP, and IGMP messages, utilizing the underlying network driver.

IP Addresses

Each computer on the Internet has a unique 32-bit identifier called an IP address. There are five classes

of IP addresses as described in Figure 5. The ranges of the five IP address classes are as follows:

Class	Range
A	0.0.0.0 to 127.255.255.255
B	128.0.0.0 to 191.255.255.255
C	192.0.0.0 to 223.255.255.255
D	224.0.0.0 to 239.255.255.255
E	240.0.0.0 to 247.255.255.255

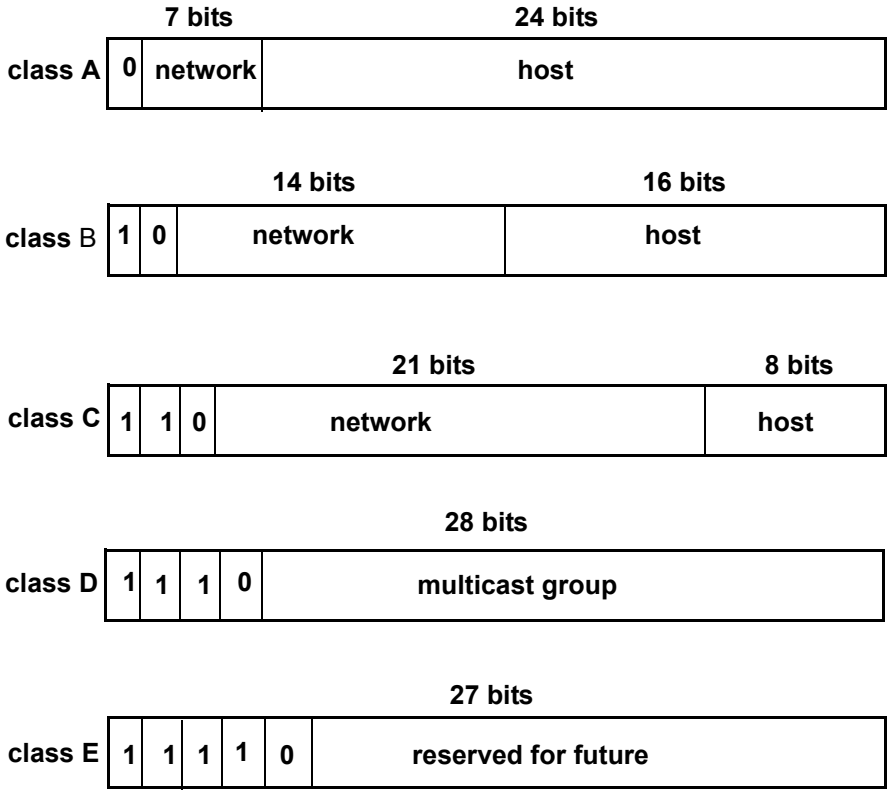

FIGURE 5. IP Address Structure

There are also three types of address specifications: *unicast*, *broadcast*, and *multicast*. Unicast addresses are those IP addresses that identify a specific host on the Internet. Unicast addresses can be either a source or a destination IP address. A broadcast address identifies all hosts on a specific network or sub-network and can only be used as destination addresses. Broadcast addresses are specified by having the host ID portion of the address set to ones. Multicast addresses (Class D) specify a dynamic group of hosts on the Internet. Members of the multicast group may join and leave whenever they wish.

Only connectionless protocols like IP and UDP can utilize broadcast and the limited broadcast capability of the multicast group.

*The macro IP_ADDRESS is defined in **nx_api.h**. It allows easy specification of IP addresses using commas instead of a periods. For example, IP_ADDRESS(128,0,0,0) specifies the first class B address shown in Figure 5.*

Gateway IP Address

In addition to the different types of network, loopback, and broadcast addresses, it is possible to set the IP instance gateway IP address using the ***nx_ip_gateway_address_set*** service. A gateway resides on the local network, and its purpose is to provide a place (“next hop”) to transmit packets whose destination lies outside the local network. Once set, all out-of network requests are routed by NetX to the gateway. Note that the default gateway must be directly accessible through one of the physical interfaces.

IP Header

For any packet to be sent on the Internet, it must have an IP header. When higher-level protocols

(UDP, TCP, ICMP, or IGMP) call the IP component to send a packet, an IP header is placed in front of the beginning of the packet. Conversely, when IP packets are received from the network, the IP component removes the IP header from the packet before delivery to the higher-level protocols. Figure 6 shows the format of the IP header.

*All headers in the TCP/IP implementation are expected to be in **big endian** format. In this format, the most significant byte of the word resides at the lowest byte address. For example, the 4-bit version and the 4-bit header length of the IP header must be located on the first byte of the header.*

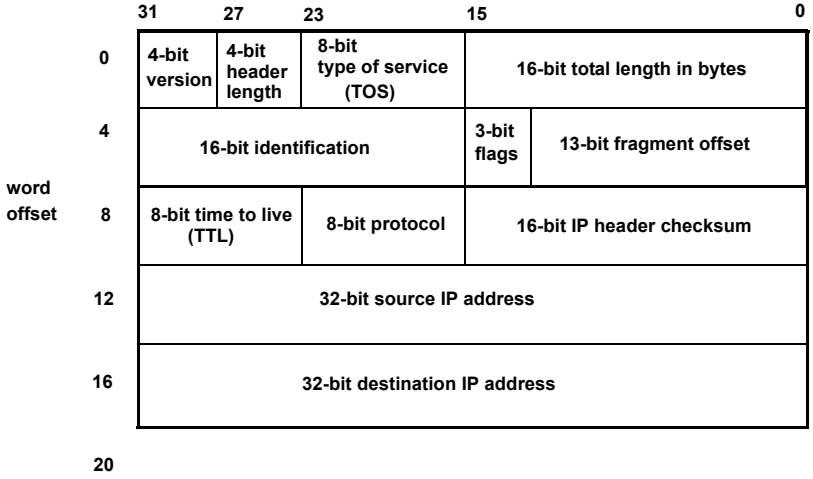

FIGURE 6. IP Header Format

The fields of the IP header are defined as follows:

IP Header Field	Purpose
4-bit version	This field contains the version of IP this header represents. For IP version 4, which is what NetX supports, the value of this field is 4.
4-bit header length	This field specifies the number of 32-bit words in the IP header. If no option words are present, the value for this field is 5.
8-bit type of service (TOS)	This field specifies the type of service requested for this IP packet. Valid requests are as follows:

TOS Request	Value
Normal	0x00
Minimum Delay	0x10
Maximum Data	0x08
Maximum Reliability	0x04
Minimum Cost	0x02

16-bit total length	This field contains the total length of the IP datagram in bytes–including the IP header. An IP datagram is the basic unit of information found on a TCP/IP Internet. It contains a destination and source address in addition to data. Because it is a 16-bit field, the maximum size of an IP datagram is 65,535 bytes.
16-bit identification	The field is a number used to uniquely identify each IP datagram sent from a host. This number is typically incremented after an IP datagram is sent. It is especially useful in un-fragmenting received IP packets.
3-bit flags	This field contains IP fragmentation information. Bit 14 is the “don’t fragment” bit. If this bit is set, the outgoing IP datagram will not be fragmented. Bit 13 is the “more fragments” bit, which is used to fragment or un-fragment IP datagrams. If this bit is set, there are more fragments. If this bit is clear, this is the last fragment of the IP packet.

IP Header Field	Purpose
13-bit fragment offset	This field contains the upper 13-bits of the fragment offset. Because of this, fragment offsets are only allowed on 8-byte boundaries. The first fragment of a fragmented IP datagram will have the “more fragments” bit set and have an offset of 0.
8-bit time to live (TTL)	This field contains the number of routers this datagram can pass, which basically limits the lifetime of the datagram.
8-bit protocol	This field specifies which protocol is using the IP datagram. The following is a list of valid protocols and their values:

Protocol	Value
ICMP	0x01
IGMP	0x02
TCP	0X06
UDP	0X11

16-bit checksum	This field contains the 16-bit checksum that covers the IP header only. There are additional checksums in the higher level protocols that cover the IP payload.
32-bit source IP address	This field contains the IP address of the sender and is always a host address.
32-bit destination IP address	This field contains the IP address of the receiver or receivers if the address is a broadcast or multicast address.

IP Fragmentation

The network driver may have limits on the size of outgoing packets. This physical limit is called the maximum transmission unit (MTU). The ***nx_interface_ip_mtu_size*** member for the interface control block contains the MTU, which is initially set up by the application’s network driver during initialization.

Although not recommended, the application may generate datagrams larger than the underlying network driver's MTU size. Before transmitting such IP datagram, the IP layer must fragment such packets. At the IP receiving end, the IP layer must collect and reassemble all the fragments before sending the packet to upper layer applications. In order to support IP fragmentation and reassembly operation, the system designer must enable the IP fragmentation feature in NetX. If this feature is not enabled, incoming fragmented IP packets are discarded, as well as packets that exceed the network driver's MTU.

*IP Fragmentation can be disabled completely by defining **NX_DISABLE_FRAGMENTATION** when building the NetX library. Doing so helps reduce the code size of NetX.*

IP Send

The IP send processing in NetX is very streamlined. The prepend pointer in the packet is moved backwards to accommodate the IP header. The IP header is completed (with all the TOS, TTL, protocol, and other options specified by the calling protocol layer), the IP checksum is computed in-line, and the packet is dispatched to the associated network driver.

The IP send processing also handles initiating ARP requests if physical mapping is needed for the destination IP address. In addition, outgoing fragmentation is also coordinated from within the IP send processing.

Packets that require IP address resolution (i.e., physical mapping) are enqueued on the ARP queue until the number of packets queued exceeds the ARP queue depth (NX_ARP_MAX_QUEUE_DEPTH). If the queue depth is reached, NetX will remove the oldest packet on the queue and continue waiting for

address resolution for the remaining packets enqueued.

For hosts with multiple interfaces, NetX determines which interface the packet should be transmitted based on the packet interface specified. If the packet interface is not specified, the packet is dropped.

IP Receive

The IP receive processing is either called from the network driver or the internal IP thread (for processing the deferred queue). The IP receive processing examines the protocol field and attempts to dispatch the packet to the proper protocol component. Before the packet is actually dispatched, the IP header is removed by advancing the prepend pointer past the IP header.

IP receive processing also detects fragmented IP packets and performs the necessary steps to re-assemble them if fragmentation is enabled.

NetX determines the appropriate interface based on the interface specified in the packet. If the packet interface is NULL, NetX defaults the interface to the primary interface. This is done to guarantee compatibility with legacy NetX Ethernet drivers.

Raw IP Send

The application may send raw IP packets (packets with only an IP header and payload) directly using the ***nx_ip_raw_packet_send*** service if raw IP packet processing has been enabled with the ***nx_ip_raw_packet_enabled*** service. When transmitting a unicast packet on a multihome device, NetX will automatically determine the correct physical interface to send the packets out on based on the destination address. However, for broadcast or multicast destination addresses, the host application must use the ***nx_ip_raw_packet_interface_send***

service to explicitly set the network interface to send the packet out on.

Raw IP Receive

If raw IP packet processing is enabled, the application may receive raw IP packets through the ***nx_ip_raw_packet_receive*** service. All incoming packets are processed according to the protocol specified in the IP header. If the protocol specifies UDP, TCP, IGMP or ICMP, NetX will process the packet using the appropriate handler for the packet protocol type. If the protocol is not one of these protocols, and raw IP receive is enabled, the packet will be processed by ***nx_ip_raw_packet_receive***. In addition, application threads may suspend with an optional timeout while waiting for a raw IP packet.

Creating IP Instances

IP instances are created either during initialization or during runtime by application threads. The initial IP address, network mask, default packet pool, media driver, and memory and priority of the internal IP thread are defined by the ***nx_ip_create*** service. NetX also supports multiple interfaces within one IP instance. Therefore, it is not necessary to create an IP instance for each interface unless special circumstances arise.

A multihome host application wishing to associate all network interfaces with the IP instance does so by attaching secondary interfaces to the IP instance using the ***nx_ip_interface_attach*** service. This services stores information about the network interface (IP address, network mask) in the interface control block. This information enables NetX to determine which interface a packet has been received on and which interface a packet should be sent out. Note an IP instance must already be created before attaching any interfaces.

More details on the NetX multihome support are available in “Multiple Network Interface (Multihome) Support ” on page 69.

Default Packet Pool

Each IP instance is given a default packet pool during creation. This packet pool is used to allocate packets for ARP, RARP, ICMP, IGMP, and various TCP ACK and state changes. If the default packet pool is empty, the underlying NetX activity aborts the packet pool entirely, and returns an error message if possible.

IP Helper Thread

Each IP instance has a helper thread. This thread is responsible for handling all deferred packet processing and all periodic processing. The IP helper thread is created in ***nx_ip_create.*** This is where the thread is given its stack and priority. Note that the first processing in the IP helper thread is to finish the network driver initialization associated with the IP create service. After the network driver initialization is complete, the helper thread starts an endless loop to process packet and periodic requests.

If unexplained behavior is seen within the IP helper thread, increasing its stack size during the IP create service is the first debugging step. If the stack is too small, the IP helper thread could possibly be overwriting memory, which may cause unusual problems.

Thread Suspension

Application threads can suspend while attempting to receive raw IP packets. After a raw packet is received, the new packet is given to the first thread suspended and that thread is resumed. NetX services for receiving packets all have an optional suspension timeout. When a packet is received or the

timeout expires, the application thread is resumed with the appropriate completion status.

IP Statistics and Errors

If enabled, the NetX IP software keeps track of several statistics and errors that may be useful to the application. The following statistics and error reports are maintained for each IP instance:

Total IP Packets Sent
Total IP Bytes Sent
Total IP Packets Received
Total IP Bytes Received
Total IP Invalid Packets
Total IP Receive Packets Dropped
Total IP Receive Checksum Errors
Total IP Send Packets Dropped
Total IP Fragments Sent
Total IP Fragments Received

All of these statistics and error reports are available to the application with the ***nx_ip_info_get*** service.

IP Control Block NX_IP

The characteristics of each IP instance are found in its control block. It contains useful information such as the IP address, network mask, and the linked list of destination IP and physical hardware address mapping. This structure is defined in the ***nx_api.h*** file. It also contains an array of network interfaces, the number of which is specified by the user configurable option NX_MAX_PHYSICAL_INTERFACES. The default value is 1 indicating a single interface device.

IP instance control blocks can be located anywhere in memory, but it is most common to make the control block a global structure by defining it outside the scope of any function.

Multiple Network Interface (Multihome) Support

NetX supports hosts connected to multiple physical network interfaces using a single IP instance. To utilize multihome support, set the user configurable option NX_MAX_PHYSICAL_INTERFACES to the number of physical interfaces needed.

To utilize a logical loopback interface, ensure the configurable option NX_DISABLE_LOOPBACK_INTERFACE is not set. When the loopback interface is enabled (the default), the total number of interfaces defined by NX_MAX_IP_INTERFACES is automatically updated to NX_MAX_PHYSICAL_INTERFACES + 1.

The host application creates a single IP instance for the primary interface using the ***nx_ip_create*** service. For each additional interface, the host application attaches the interface to the IP instance using the ***nx_ip_interface_attach*** service.

Each interface structure contains a subset of network information about the network interface that is contained in the IP control block, including network address, host IP address, network mask, MTU size, and Ethernet driver association. This is necessary for compatibility with legacy NetX applications.

Existing single interface NetX applications and network drivers do not require any changes when using NetX with multihomed support.

The primary interface has index zero in the IP instance list (array). Each subsequent interface attached to the IP instance is assigned the next index.

All upper layer protocol services for which the IP instance is enabled, including TCP, UDP and IGMP, are available to all the attached interfaces. These upper layer protocols are for the most part not directly

involved with choosing or determining the packet interface when sending or receiving packets.

In most cases, NetX can determine the correct interface to send the packet out on from the packet destination IP address or associated socket interface. When multiple interfaces are used, the host application must use interface-specific NetX services to explicitly set which interface broadcast and multicast packets are sent out on.

Services specifically for developing multihome applications include the following:

nx_igmp_multicast_interface_join
nx_ip_interface_address_get
nx_ip_interface_address_set
nx_ip_interface_attach
nx_ip_interface_info_get
nx_ip_interface_status_check
nx_ip_raw_packet_interface_send
nx_udp_socket_interface_send

These services are explained in greater detail in “Description of NetX Services” on page 105.

For multicast or broadcast packets, NetX has no way of knowing which interface to choose in a multihome host. If the sending component does not specify an interface, NetX defaults to the primary interface. To send a loopback packet out the logical interface, NetX library must be built without NX_DISABLE_LOOPBACK_INTERFACE being set. Nearly all NetX components (raw IP, ICMP, TCP, and UDP) follow this process.

Static IP Routing

The static routing feature supported in NetX allows a host application to specify an interface and next hop address for specific out of network destination IP addresses. If static routing is enabled, NetX

searches through the static routing table for an entry matching the destination address of the packet to send. If no match is found, NetX searches through the list of physical interfaces and chooses an interface and next hop based on the destination IP address and network mask. If the destination does not match any of the network interfaces attached to the IP instance, NetX chooses the IP instance default gateway.

Entries can be added and removed from the static routing table using the ***nx_ip_static_route_add*** and ***nx_ip_static_route_delete*** services, respectively. To use static routing, the host application must enable this feature by defining NX_ENABLE_IP_STATIC_ROUTING.

When adding an entry to the static routing table, NetX checks for a matching entry for the specified destination address already in the table. If one exists, it gives preference to the entry with the smaller network (larger number of most significant bits) in the network mask.

Address Resolution Protocol (ARP)

The Address Resolution Protocol (ARP) is responsible for dynamically mapping 32-bit IP addresses to those of the underlying physical media (RFC 826). Ethernet is the most typical physical media, and it supports 48-bit addresses. The need for ARP is determined by the IP network driver supplied to the ***nx_ip_create*** service. If physical mapping is required, the network driver must set the ***nx_interface_address_mapping_needed*** member of the associated NX_INTERFACE structure.

ARP Enable

For ARP to function properly, it must first be enabled by the application with the ***nx_arp_enable*** service. This service sets up various data structures for ARP processing, including the creation of an ARP cache area from the memory supplied to the ARP enable service.

ARP Cache

The ARP cache can be viewed as an array of internal ARP mapping data structures. Each internal structure is capable of maintaining the relationship between an IP address and a physical hardware address. In addition, each data structure has link pointers so it can be part of multiple linked lists.

ARP Dynamic Entries

By default, the ARP enable service places all entries in the ARP cache on the list of available dynamic ARP entries. A dynamic ARP entry is allocated from this list by the IP software when a send request to an unmapped IP address is detected. After allocation, the ARP entry is set up and an ARP request is sent to the physical media.

If all dynamic ARP entries are in service, the ARP entry in service the least is used for the latest mapping request.

ARP Static Entries

The application can also set up static ARP mapping by using the ***nx_arp_static_entry_create*** service. This service allocates an ARP entry from the dynamic ARP entry list and places it on the static list with the mapping information supplied by the application. Static ARP entries are not subject to reuse or aging.

ARP Messages

As mentioned previously, an ARP request message is sent when the IP software detects that mapping is needed for an IP address. ARP requests are sent periodically (every **NX_ARP_UPDATE_RATE** seconds) until a corresponding ARP response is received. A total of **NX_ARP_MAXIMUM_RETRIES** ARP requests are made before the ARP attempt is abandoned. When an ARP response is received, the associated physical address information is stored in the ARP entry that is in the cache.

For multihome applications, NetX determines which interface to send the ARP requests and responses based on the specified packet interface.

Outgoing IP packets are queued while NetX waits for the ARP response. The number of outgoing IP packets queued is defined by the constant NX_ARP_MAX_QUEUE_DEPTH.

NetX also responds to ARP requests from other nodes on the local network. When an external ARP request is made that matches the current IP address, NetX builds an ARP response message that contains the current physical address.

The formats of Ethernet ARP requests and responses are shown in Figure 7 on page 76 and are described below.

Request/Response Field	Purpose
Ethernet Destination Address	This 6-byte field contains the destination address for the ARP response and is a broadcast (all ones) for ARP requests. This field is setup by the network driver.
Ethernet Source Address	This 6-byte field contains the address of the sender of the ARP request or response and is set up by the network driver.

Request/Response Field	Purpose
Frame Type	This 2-byte field contains the type of Ethernet frame present and, for ARP requests and responses, this is equal to 0x0806. This is the last field the network driver is responsible for setting up.
Hardware Type	This 2-byte field contains the hardware type, which is 0x0001 for Ethernet.
Protocol Type	This 2-byte field contains the protocol type, which is 0x0800 for IP addresses.
Hardware Size	This 1-byte field contains the hardware address size, which is 6 for Ethernet addresses.
Protocol Size	This 1-byte field contains the IP address size, which is 4 for IP addresses.
Operation Code	This 2-byte field contains the operation for this ARP packet. An ARP request is specified with the value of 0x0001, while an ARP response is represented by a value of 0x0002.
Sender Ethernet Address	This 6-byte field contains the sender's Ethernet address.
Sender IP Address	This 4-byte field contains the sender's IP address.
Target Ethernet Address	This 6-byte field contains the target's Ethernet address.
Target IP Address	This 4-byte field contains the target's IP address.

ARP requests and responses are Ethernet-level packets. All other TCP/IP packets are encapsulated by an IP packet header.

*All ARP messages in the TCP/IP implementation are expected to be in **big endian** format. In this format, the most significant byte of the word resides at the lowest byte address.*

ARP Aging

Automatic invalidation of dynamic ARP entries is supported. The constant

NX_ARP_EXPIRATION_RATE specifies the number of seconds an established IP address to physical mapping stays valid. After expiration, the ARP entry is removed from the ARP cache. The next attempt to send to the corresponding IP address will result in a new ARP request. ARP aging is disabled by default. The default value is zero for the **NX_ARP_EXPIRATION_RATE** constant.

ARP Statistics and Errors

If enabled, the NetX ARP software keeps track of several statistics and errors that may be useful to the application. The following statistics and error reports are maintained for each IP's ARP processing:

Total ARP Requests Sent
Total ARP Requests Received
Total ARP Responses Sent
Total ARP Responses Received
Total ARP Dynamic Entries
Total ARP Static Entries
Total ARP Aged Entries
Total ARP Invalid Messages

All these statistics and error reports are available to the application with the ***nx_arp_info_get*** service.

Reverse Address Resolution Protocol (RARP)

The Reverse Address Resolution Protocol (RARP) is the protocol for requesting network assignment of the host's 32-bit IP addresses (RFC 903). This is done through an RARP request and continues periodically until a network member assigns an IP address to the host network interface in an RARP response. The IP create service ***nx_ip_create*** service (and ***nx_ip_interface_attach*** service for multihome hosts)

create a need for RARP by supplying a zero IP address. If RARP is enabled by the host application, it can use the RARP protocol to request an IP address from the network server for each network interface with a zero IP address.

RARP Enable

To use RARP, the application must create the IP instance with an IP address of zero, then enable RARP. For multihome hosts, at least one interface associated with the IP instance must have an IP address of zero. The RARP processing periodically sends RARP request messages for each network interface requiring an IP address until a valid RARP reply with the network designated IP address for that interface is received. At this point, RARP processing is complete.

RARP Request

The format of an RARP request packet is almost identical to the ARP packet shown in Figure 7. The

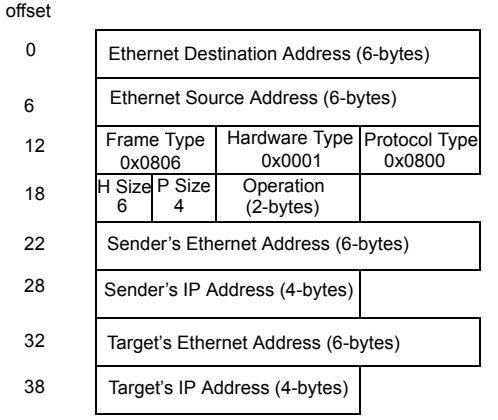

FIGURE 7. ARP Packet Format

only difference is the frame type field is 0x8035 and the *Operation Code* field is 3, designating an RARP request. As mentioned previously, RARP requests will be sent periodically (every **NX_RARP_UPDATE_RATE** seconds) until a RARP reply with the network assigned IP address is received.

*All RARP messages in the TCP/IP implementation are expected to be in **big endian** format. In this format, the most significant byte of the word resides at the lowest byte address.*

RARP Reply

RARP reply messages are received from the network and contain the network assigned IP address for this host. The format of an RARP reply packet is almost identical to the ARP packet shown in Figure 7. The only difference is the frame type field is 0x8035 and the *Operation Code* field is 4, which designates an RARP reply. After received, the IP address is setup in the IP instance, the periodic RARP request is disabled, and the IP instance is now ready for normal network operation.

For multihome hosts, the IP address is applied to the requesting network interface. If there are other network interfaces still requesting an IP address assignment, the periodic RARP service continues until all interface IP address requests are resolved.

*The application should not use the IP instance until the RARP processing is complete. The **nx_ip_status_check** may be used by threads to wait for the RARP completion. For multihome hosts, the application should not use the requesting interface until the RARP processing is complete on that interface. Secondary interface IP address status can be checked with the **nx_ip_interface_status_check** service.*

RARP Statistics and Errors

If enabled, the NetX RARP software keeps track of several statistics and errors that may be useful to the application. The following statistics and error reports are maintained for each IP's RARP processing:

Total RARP Requests Sent
Total RARP Responses Received
Total RARP Invalid Messages

All these statistics and error reports are available to the application with the ***nx_rarp_info_get*** service.

Internet Control Message Protocol (ICMP)

The Internet Control Message Protocol (ICMP) is responsible for passing error and control information between IP network members (RFC 792). Like most other TCP/IP messages, ICMP messages are encapsulated by an IP header with the ICMP protocol designation.

ICMP Enable

Before ICMP messages can be processed by NetX, the application must call the ***nx_icmp_enable*** service to enable ICMP processing. After this is done, the application can issue ping requests and field ping responses.

Ping Request

A ping request is one type of ICMP message that is typically used to check for the existence of a specific member on the network, as identified by a host IP address. If the specific host is present, its ICMP component processes the ping request by issuing a

ping response. Figure 8 details the ICMP ping message format.

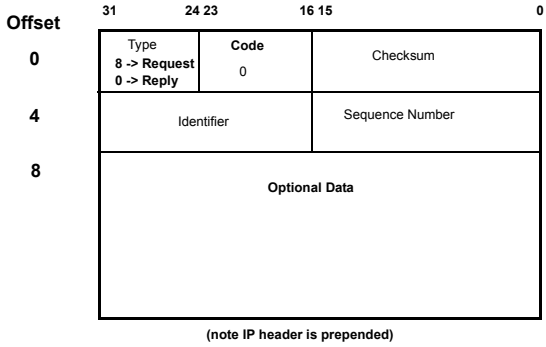

FIGURE 8. ICMP Ping Message

*All ICMP messages in the TCP/IP implementation are expected to be in **big endian** format. In this format, the most significant byte of the word resides at the lowest byte address.*

The following describes the ICMP header format:

Header Field	***Purpose***
Type	This field specifies the ICMP message (bits 31-28). The most common are: 0 Echo Reply 3 Destination Unreachable 8 Echo Request
Code	This field is context specific on the type field (bits 27-24). For an echo request or reply the code is set to zero
Checksum	This field contains the 16-bit checksum of the one's complement sum of the ICMP message including the entire the ICMP header

Identification	This field contains an ID value identifying the host; a host should use the ID extracted from an ECHO request in the ECHO REPLY (bits 31-16)
Sequence	This field contains an ID value; a host should use the ID extracted from an ECHO request in the ECHO REPLY (bits 31-16). Unlike the identifier field, this value will change in a subsequent Echo request from the same host (bits 15-0)

Ping Response

A ping response is another type of ICMP message that is generated internally by the ICMP component in response to an external ping request. In addition to acknowledgement, the ping response also contains a copy of the user data supplied in the ping request.

Thread Suspension

Application threads can suspend while attempting to ping another network member. After a ping response is received, the ping response message is given to the first thread suspended and that thread is resumed. Like all NetX services, suspending on a ping request has an optional timeout.

ICMP Statistics and Errors

If enabled, the NetX ICMP software keeps track of several statistics and errors that may be useful to the application. The following statistics and error reports are maintained for each IP's ICMP processing:

Total ICMP Pings Sent
Total ICMP Ping Timeouts
Total ICMP Ping Threads Suspended
Total ICMP Ping Responses Received
Total ICMP Checksum Errors
Total ICMP Unhandled Messages
Total ICMP Pings Received
Total ICMP Pings Responsed To

All these statistics and error reports are available to the application with the ***nx_icmp_info_get*** service.

Internet Group Management Protocol (IGMP)

The Internet Group Management Protocol (IGMP) provides UDP packet delivery to multiple network members that belong to the same multicast group (RFC 1112 and RFC 2236). A multicast group is basically a dynamic collection of network members and is represented by a Class D IP address. Members of the multicast group may leave at any time, and new members may join at any time. The coordination involved in joining and leaving the group is the responsibility of IGMP.

IGMP Enable

Before any multicasting activity can take place in NetX, the application must call the ***nx_igmp_enable*** service. This service performs basic IGMP initialization in preparation for multicast requests.

Multicast IP Addresses

As mentioned previously, multicast addresses are actually Class D IP addresses as shown in Figure 5 on page 59. The lower 28-bits of the Class D address correspond to the multicast group ID. There are a series of pre-defined multicast addresses; however, the *all hosts address* (244.0.0.1) is particularly important to IGMP processing. The *all hosts address* is used by routers to query all multicast members to report on which multicast groups they belong to.

Physical Address Mapping

Class D multicast addresses map directly to physical Ethernet addresses ranging from 01.00.5e.00.00.00 through 01.00.5e.7f.ff.ff. The lower 23 bits of the IP multicast address map directly to the lower 23 bits of the Ethernet address.

Multicast Group Join

Applications that need to join a particular multicast group may do so by calling the ***nx_igmp_multicast_join*** service. This service keeps track of the number of requests to join this multicast group. If this is the first application request to join the multicast group, an IGMP report is sent out on the network indicating this host's intention to join the group. Next, the network driver is called to set up for listening for packets with the Ethernet address for this multicast group.

For multihome hosts, the ***nx_igmp_multicast_interface_join*** service should be used instead of *nx_igmp_multicast_join,* if the multicast group destination address is on a secondary network interface. The original service *nx_igmp_multicast_join* service is limited to multicast groups on the primary network and is included for backward compatibility.

Multicast Group Leave

Applications that need to leave a previously joined multicast group may do so by calling the ***nx_igmp_multicast_leave*** service. This service reduces the internal count associated with how many times the group was joined. If there are no outstanding join requests for a group, the network driver is called to disable listening for packets with this multicast group's Ethernet address.

IGMP Report Message

When the application joins a multicast group, an IGMP report message is sent via the network to indicate the host's intention to join a particular multicast group. The format of the IGMP report message is shown in Figure 9. The multicast group address is used for both the group message in the IGMP report message and the destination IP address.

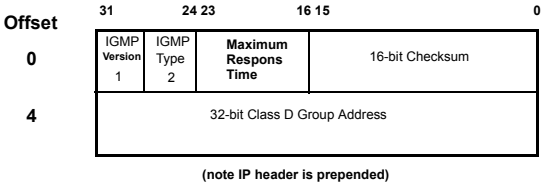

FIGURE 9. IGMP Report Message

In the figure above, the IGMP header contains a version field, a type field, a checksum field, and a multicast group address field. For IGMPv1 messages, the Maximum Response Time field is always set to zero, as this is not part of the IGMPv1 protocol. The Maximum Response Time field is set when the host receives a Query type IGMP message and cleared when a host receives another hosts Report type message as defined by the IGMPv2 protocol.

The following describes the IGMP header format:

Header Field	*Purpose*
Version	This field specifies the IGMP version (bits 31- 28)
Type	This field specifies the type of IGMP message (bits 27 -24)
Identifier	Not used in IGMP v1. In IGMP v2 this field serves as the maximum response time.

Checksum This field contains the 16-bit checksum of the one's complement sum of the IGMP message starting with the IGMP version (bits 0-15)

Group Address 32-bit class D group IP address

IGMP report messages are also sent in response to IGMP query messages sent by a multicast router. Multicast routers periodically send query messages out to see which hosts still require group membership. Query messages have the same format as the IGMP report message shown in Figure 9. The only differences are the IGMP type is equal to 1 and the group address field is set to 0. IGMP Query messages are sent to the *all hosts* IP address by the multicast router. A host that still wishes to maintain group membership responds by sending another IGMP report message.

*All messages in the TCP/IP implementation are expected to be in **big endian** format. In this format, the most significant byte of the word resides at the lowest byte address.*

IGMP Statistics and Errors

If enabled, the NetX IGMP software keeps track of several statistics and errors that may be useful to the application. The following statistics and error reports are maintained for each IP's IGMP processing:

Total IGMP Reports Sent
Total IGMP Queries Received
Total IGMP Checksum Errors
Total IGMP Current Groups Joined

All these statistics and error reports are available to the application with the ***nx_igmp_info_get*** service.

User Datagram Protocol (UDP)

The User Datagram Protocol (UDP) provides the simplest form of data transfer between network members (RFC 768). UDP data packets are sent from one network member to another in a best effort fashion; i.e., there is no built-in mechanism for acknowledgement by the packet recipient. In addition, sending a UDP packet does not require any connection to be established in advance. Because of this, UDP packet transmission is very efficient.

UDP Enable

Before UDP packet transmission is possible, the application must first enable UDP by calling the ***nx_udp_enable*** service. After enabled, the application is free to send and receive UDP packets.

UDP Header

UDP places a simple packet header in front of the application's data when sending application data and removes a similar UDP header from the packet before delivering a received UDP packet to the application. UDP utilizes the IP protocol for sending and receiving packets, which means there is an IP header in front of the UDP header when the packet is on the network. Figure 10 shows the format of the UDP header.

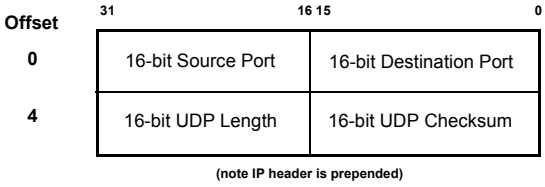

FIGURE 10. UDP Header

*All headers in the TCP/IP implementation are expected to be in **big endian** format. In this format,*

the most significant byte of the word resides at the lowest byte address.

The following describes the UDP header format:

Header Field	Purpose
16-bit source port number	This field contains the port on which the UDP packet is being sent. Valid UDP ports range from 1 through 0xFFFF.
16-bit destination port number	This field contains the UDP port to which the packet is being sent. Valid UDP ports range from 1 through 0xFFFF.
16-bit UDP length	This field contains the number of bytes in the UDP packet, including the size of the UDP header.
16-bit UDP checksum	This field contains the 16-bit checksum for the packet, including the UDP header, the packet data area, and the pseudo IP header.

UDP Checksum

UDP specifies a one's complement 16-bit checksum that covers the IP pseudo header (consisting of the 32-bit source IP address, 32-bit destination IP address, and the protocol/length IP word), the UDP header, and the UDP packet data. If the calculated UPD checksum is 0, it is stored as all ones (0xFFFF). If the sending socket has the UDP checksum logic disabled, a zero is placed in the UDP checksum field to indicate the checksum was not calculated.

If the UDP checksum does not match the computed checksum by the receiver, the UDP packet is simply discarded.

NetX allows the application to enable or disable UDP checksum calculation on a per-socket basis. By

default, the UDP socket checksum logic is enabled. The application can disable checksum logic for a particular UDP socket by calling the ***nx_udp_socket_checksum_disable***.

Disabling the UDP checksum logic is appropriate if the application is performing its own checksum logic on the data packet or the local network driver has sufficient error detection logic.

UDP Ports and Binding

A UDP port is a logical end point in the UDP protocol. There are 65,535 valid ports in the UDP component of NetX, ranging from 1 through 0xFFFF. To send or receive UDP data, the application must first create a UDP socket, then bind it to a desired port. After bound to a port, the application may send and receive data on that socket.

UDP Fast Path™

The UDP Fast Path™ is the name for a low packet overhead path through the NetX UDP implementation. Sending a UDP packet requires just three function calls: ***nx_udp_socket_send***, ***nx_ip_socket_send***, and the eventual call to the network driver. On UDP packet reception, the UDP packet is either placed on the appropriate UDP socket receive queue or delivered to a suspended application thread in a single function call from the network driver's receive interrupt processing. This highly optimized logic for sending and receiving UDP packets is the essence of UDP Fast Path technology.

UDP Packet Send

Sending UDP data is easily accomplished by calling the ***nx_udp_socket_send*** function. This service places a UDP header in front of the packet and sends it on the Internet using the internal IP send routine. There is no thread suspension on sending UDP

packets because all UDP packet transmissions are processed immediately.

For multihome devices, **nx_udp_socket_send** will work with packets whose destination address is on either the primary or secondary network interface. NetX can figure out from the destination IP address the correct interface and next hop. However, for broadcast packets that must go out a secondary interface, the host application must explicitly set which interface to use by calling **nx_udp_socket_interface_send**.

If UDP checksum logic is enabled for this socket, the checksum operation is performed in the context of the calling thread, without blocking access to the UDP or IP data structures.

The UDP data residing in the NX_PACKET structure should reside on a long-word boundary. There also needs to be sufficient space between the prepend pointer and the data start pointer to place the UDP, IP, and physical media headers.

UDP Packet Receive

Application threads may receive UDP packets from a particular socket by calling ***nx_udp_socket_receive***. The socket receive function delivers the oldest packet on the socket's receive queue. If there are no packets on the receive queue, the calling thread can suspend (with an optional timeout) until a packet arrives.

The UDP receive packet processing (usually called from the network driver's receive interrupt handler) is responsible for either placing the packet on the UDP socket's receive queue or delivering it to the first suspended thread waiting for a packet. If the packet is queued, the receive processing also checks the maximum receive queue depth associated with the

socket. If this newly queued packet exceeds the queue depth, the oldest packet in the queue is discarded.

UDP Receive Notify

If the application thread needs to process received data from more than one socket, the ***nx_udp_socket_receive_notify*** function should be used. This function registers a receive packet callback function for the socket. Whenever a packet is received on the socket, the callback function is executed.

The contents of the callback function is application-specific; however, it would most likely contain logic to inform the processing thread that a packet is now available on the corresponding socket.

UDP Socket Create

UDP sockets are created either during initialization or during runtime by application threads. The initial type of service, time to live, and receive queue depth are defined by the ***nx_udp_socket_create*** service. There are no limits on the number of UDP sockets in an application.

Thread Suspension

As mentioned previously, application threads can suspend while attempting to receive a UDP packet on a particular UDP port. After a packet is received on that port, it is given to the first thread suspended and that thread is then resumed. An optional timeout is available when suspending on a UDP receive packet, a feature available for most NetX services.

UDP Socket Statistics and Errors

If enabled, the NetX UDP socket software keeps track of several statistics and errors that may be useful to the application. The following statistics and

error reports are maintained for each IP/UDP instance:

Total UDP Packets Sent
Total UDP Bytes Sent
Total UDP Packets Received
Total UDP Bytes Received
Total UDP Invalid Packets
Total UDP Receive Packets Dropped
Total UDP Receive Checksum Errors
UDP Socket Packets Sent
UDP Socket Bytes Sent
UDP Socket Packets Received
UDP Socket Bytes Received
UDP Socket Packets Queued
UDP Socket Receive Packets Dropped
UDP Socket Checksum Errors

All these statistics and error reports are available to the application with the ***nx_udp_info_get*** and ***nx_udp_socket_info_get*** services.

UDP Socket Control Block TX_UDP_SOCKET

The characteristics of each UDP socket are found in the associated NX_UDP_SOCKET control block. It contains useful information such as the link to the IP data structure, the network interface for the sending and receiving paths, the bound port, and the receive packet queue. This structure is defined in the ***nx_api.h*** file.

Transmission Control Protocol (TCP)

The Transmission Control Protocol (TCP) provides reliable stream data transfer between two network members (RFC 793). All data sent from one network member are verified and acknowledged by the receiving member. In addition, the two members must have established a connection prior to any data transfer. All this results in reliable data transfer; however, it does require substantially more overhead than the previously described UDP data transfer.

TCP Enable

Before TCP connections and packet transmissions are possible, the application must first enable TCP by calling the ***nx_tcp_enable*** service. After enabled, the application is free to access all TCP services.

TCP Header

TCP places a somewhat complex packet header in front of the application's data when sending data and removes a similar TCP header from the packet before delivering a received TCP packet to the application. TCP utilizes the IP protocol to send and receive packets, which means there is an IP header in front of the TCP header when the packet is on the network. Figure 11 shows the format of the TCP header.

The following describes the TCP header format:

Header Field	Purpose
16-bit source port number	This field contains the port the TCP packet is being sent out on. Valid TCP ports range from 1 through 0xFFFF.
16-bit destination port number	This field contains the TCP port the packet is being sent to. Valid TCP ports range from 1 through 0xFFFF.

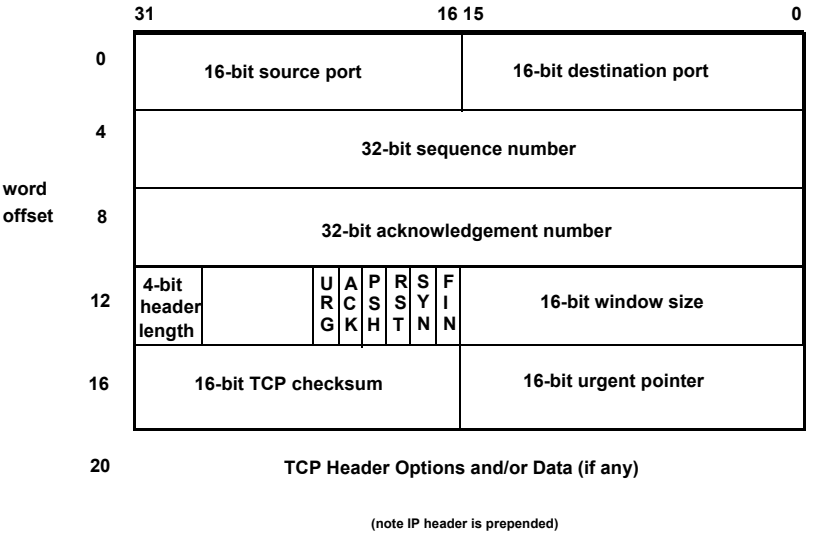

FIGURE 11. TCP Header

Header Field	Purpose
32-bit sequence number	This field contains the sequence number for data sent from this end of the connection. The original sequence is established during the initial connection sequence between two TCP nodes. Every data transfer from that point results in an increment of the sequence number by the amount bytes sent.
32-bit acknowledgement number	This field contains the sequence number corresponding to the last byte received by this side of the connection. This is used to determine whether or not data previously sent has successfully been received by the other end of the connection.
4-bit header length	This field contains the number of 32-bit words in the TCP header. If no options are present in the TCP header, this field is 5.

Header Field	Purpose
6-bit code bits	This field contains the six different code bits used to indicate various control information associated with the connection. The control bits are defined as follows:

Name	Bit	Meaning
URG	21	Urgent data present
ACK	20	Acknowledgement number is valid
PSH	19	Handle this data immediately
RST	18	Reset the connection
SYN	17	Synchronize sequence numbers (used to establish connection)
FIN	16	Sender is finished with transmit (used to close connection)

16-bit window	This field contains the amount of bytes the sender can currently receive. This basically is used for flow control. The sender is responsible for making sure the data to send will fit into the receiver's advertised window.
16-bit TCP checksum	This field contains the 16-bit checksum for the packet including the TCP header, the packet data area, and the pseudo IP header.
16-bit urgent pointer	This field contains the positive offset of the last byte of the urgent data. This field is only valid if the URG code bit is set in the header.

*All headers in the TCP/IP implementation are expected to be in **big endian** format. In this format, the most significant byte of the word resides at the lowest byte address.*

TCP Checksum

TCP specifies a one's complement 16-bit checksum that covers the IP pseudo header (consisting of the 32-bit source IP address, 32-bit destination IP address, and the protocol/length IP word), the TCP header, and the TCP packet data.

TCP Ports

A TCP port is a logical connection point in the TCP protocol. There are 65,535 valid ports in the TCP component of NetX, ranging from 1 through 0xFFFF. Unlike UDP in which data from one port can be sent to any other destination port, a TCP port is connected to another specific TCP port, and only when this connection is established can any data transfer take place—and only between the two ports making up the connection.

TCP ports are completely separate from UDP ports; e.g., UDP port number 1 has no relation to TCP port number 1.

Client Server Model

To use TCP for data transfer, a connection must first be established between the two TCP sockets. The establishment of the connection is done in a client-server fashion. The client side of the connection is the side that initiates the connection, while the server side simply waits for client connection requests before any processing is done.

For multihome devices, NetX automatically determines the interface and next hop address on the client side for transmitting packets based on the packet destination IP address. Because TCP does not operate on broadcast addresses, there is no need to pass in a "hint" for the outgoing interface.

TCP Socket State Machine

The connection between two TCP sockets (one client and one server) is complex and is managed in a state machine manner. Each TCP socket starts in a CLOSED state. Through connection events each socket's state machine migrates into the ESTABLISHED state, which is where the bulk of the data transfer in TCP takes place. When one side of the connection no longer wishes to send data, it disconnects, and this action eventually causes both

TCP sockets to return to the CLOSED state. This process repeats each time a TCP client and server establish and close a connection. Figure 12 on page 96 shows the various states of the TCP state machine.

TCP Client Connection

As mentioned previously, the client side of the TCP connection initiates a connection request to a TCP server. Before a connection request can be made, TCP must be enabled on the client IP instance. In addition, the client TCP socket must next be created with ***nx_tcp_socket_create*** service and bound to a port via the ***nx_tcp_client_socket_bind*** service.

After the client socket is bound, the ***nx_tcp_client_socket_connect*** service is used to establish a connection with a TCP server. Note the socket must be in a CLOSED state to initiate a connection attempt. Establishing the connection starts with NetX issuing a SYN packet and then waiting for a SYN ACK packet back from the server, which signifies acceptance of the connection request. After the SYN ACK is received, NetX responds with an ACK packet and promotes the client socket to the ESTABLISHED state.

For multihome hosts, NetX automatically determines which network interface the client connects to the server based on the server IP address. It saves that interface in the client TCP socket control block for subsequent packet transmissions on the same connection.

TCP Client Disconnection

Closing the connection is accomplished by calling ***nx_tcp_socket_disconnect***. If no suspension is specified, the client socket sends a RST packet to the server socket and places the socket in the CLOSED

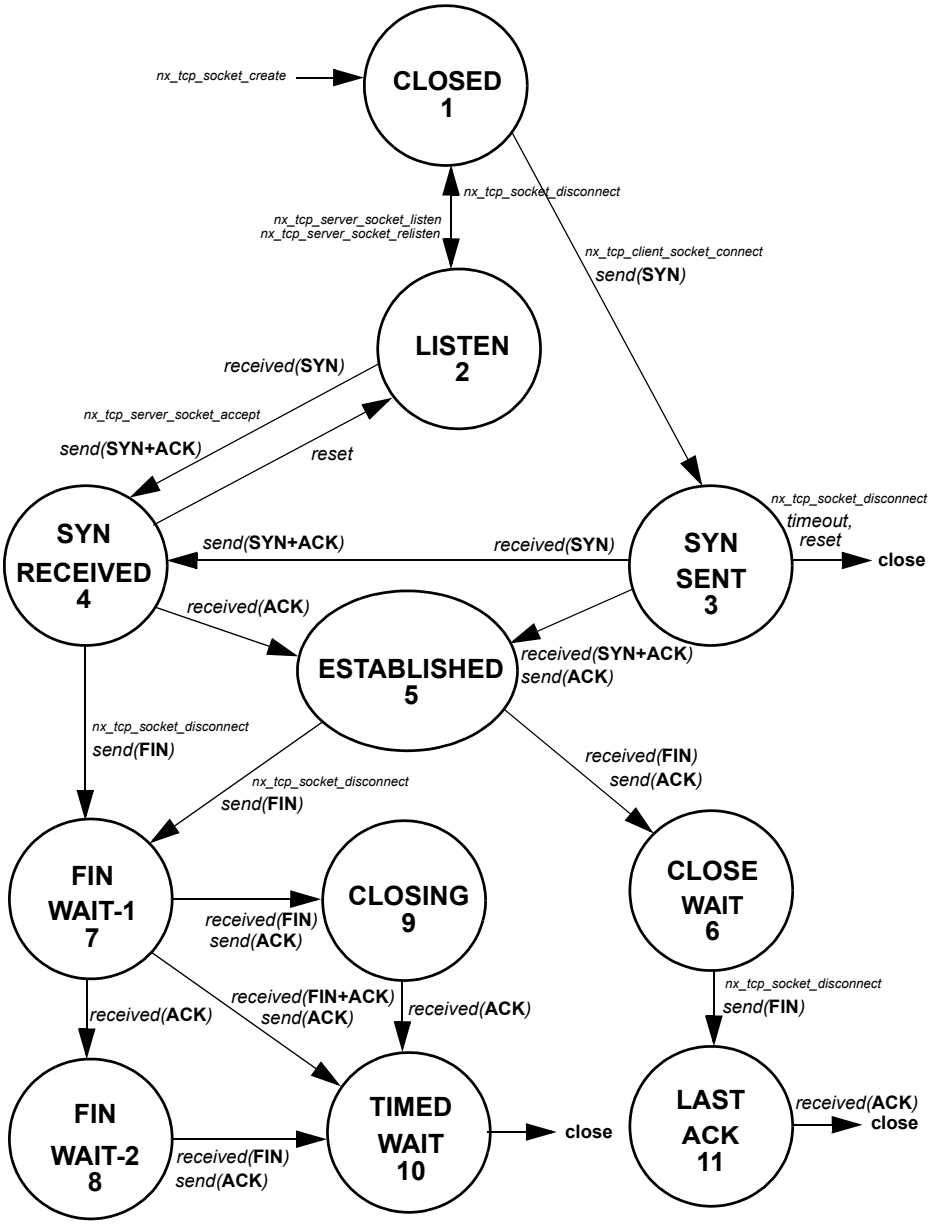

FIGURE 12. States of the TCP State Machine

state. Otherwise, if a suspension is requested, the full TCP disconnect protocol is performed, as follows:

- If the server previously initiated a disconnect request (the client socket has already received a FIN packet, responded with an ACK, and is in the CLOSE WAIT state), NetX promotes the TCP socket state to the LAST ACK state and sends a FIN packet. It then waits for an ACK from the server before completing the disconnect and entering the CLOSED state.
- If on the other hand, the client is the first to initiate a disconnect request (the server has not disconnected and the socket is still in the ESTABLISHED state), NetX sends a FIN packet to initiate the disconnect and waits to receive a FIN and an ACK from the server before completing the disconnect and placing the socket in a CLOSED state.

If there are still packets on the socket transmit queue, NetX suspends for the specified timeout to allow the packets to be acknowledged. If the timeout expires, NetX empties the transmit queue of the client socket.

To unbind the port and client socket, the application calls ***nx_tcp_client_socket_unbind***. The socket must be in a CLOSED state or in the process of disconnecting (i.e., CLOSE WAIT state) before the port is released; otherwise, an error is returned.

Finally, if the application no longer needs the client socket, it calls ***nx_tcp_socket_delete*** to delete the socket.

TCP Server Connection

The server side of a TCP connection is passive; i.e., the server waits for a client connection request. To accept a client connection, TCP must first be enabled on the IP instance. Next, the application must create a TCP socket using the ***nx_tcp_socket_create*** service.

The server socket must also be setup for listening for connection requests using the ***nx_tcp_server_socket_listen*** service. This service places the server socket in the LISTEN state and binds the specified server port to the server socket. If the socket connection has already been established, the function simply returns a successful status.

To set a socket listen callback routine the application specifies the appropriate callback function for the tcp_listen_callback argument of the ***nx_tcp_server_socket_listen*** *service. This application callback function is then executed by NetX whenever a new connection is requested on this server port. The processing in the callback is under application control.*

To accept client connection requests, the application calls the ***nx_tcp_server_socket_accept*** service. The server socket must either be in a LISTEN state or a SYN RECEIVED state (i.e., the server has received a SYN packet from a client requesting a connection) to call the accept service. A successful return status from from ***nx_tcp_server_socket_accept*** indicates the connection has been established and the server socket is in the ESTABLISHED state.

After the server socket has a valid connection, additional client connection requests are queued up to the depth specified by the ***nx_tcp_server_socket_listen*** service. In order to process subsequent connections on a server port, the application must call ***nx_tcp_server_socket_relisten*** with an available socket (i.e., a socket in a CLOSED state). Note that the same server socket could be used if the previous connection associated with the socket is now finished and the socket is in the CLOSED state.

TCP Server Disconnection

Closing the connection is accomplished by calling ***nx_tcp_socket_disconnect***. If no suspension is specified, the server socket sends a RST packet to the client socket and places the socket in the CLOSED state. Otherwise, if a suspension is requested, the full TCP disconnect protocol is performed, as follows:

- If the client previously initiated a disconnect request (the server socket has already received a FIN packet, responded with an ACK, and is in the CLOSE WAIT state), NetX promotes the TCP socket state to the LAST ACK state and sends a FIN packet. It then waits for an ACK from the client before completing the disconnect and entering the CLOSED state.
- If on the other hand, the server is the first to initiate a disconnect request (the client has not disconnected and the socket is still in the ESTABLISHED state), NetX sends a FIN packet to initiate the disconnect and waits to receive a FIN and an ACK from the client before completing the disconnect and placing the socket in a CLOSED state.

If there are still packets on the socket transmit queue, NetX suspends for the specified timeout to allow the packets to be acknowledged. If the timeout expires, NetX empties the transmit queue of the server socket.

After the disconnect processing is complete and the server socket is in the CLOSED state, the application must call the ***nx_tcp_server_socket_unaccept*** service to end the association of this socket with the server port. Note this service must be called by the application even if ***nx_tcp_socket_disconnect*** or ***nx_tcp_server_socket_accept*** return an error status. After the ***nx_tcp_server_socket_unaccept*** returns, the socket can be used as a client or server socket, or even deleted if it is no longer needed. If

accepting another client connection on the same server port is desired, the ***nx_tcp_server_socket_relisten*** service should be called with this socket.

Stop Listening on a Server Port

If the application no longer wishes to listen for client connection requests on a server port that was previously specified by a call to the ***nx_tcp_server_socket_listen*** service, the application simply calls the ***nx_tcp_server_socket_unlisten*** service. This service places any socket waiting for a connection back in the CLOSED state and releases any queued client connection request packets.

TCP Window Size

During both the setup and data transfer phases of the connection, each port reports the amount of data it can handle, which is called its window size. As data are received and processed, this window size is adjusted dynamically. In TCP, a sender can only send an amount of data that is less than or equal to the amount of data specified by the receiver's window size. In essence, the window size provides flow control for data transfer in each direction of the connection.

TCP Packet Send

Sending TCP data is easily accomplished by calling the ***nx_tcp_socket_send*** function. This service first builds a TCP header in front of the packet (including the checksum calculation). If the receiver's window size is larger than the data in this packet, the packet is sent on the Internet using the internal IP send routine. Otherwise, the caller may suspend and wait for the receiver's window size to increase enough for this packet to be sent. At any given time, only one sender may suspend while trying to send TCP data.

The TCP data residing in the NX_PACKET structure should reside on a long-word boundary. In addition, there needs to be sufficient space between the prepend pointer and the data start pointer to place the TCP, IP, and physical media headers.

TCP Packet Retransmit

TCP packets sent are actually stored internally until an ACK is returned from the other side of the connection. If an ACK is not received within the timeout period, the transmit packet is re-sent and the next timeout period is increased. When an ACK is received, all packets covered by the acknowledgement number in the internal transmit sent queue are finally released.

TCP Packet Receive

The TCP receive packet processing (called from the IP helper thread) is responsible for handling various connection and disconnection actions as well as transmit acknowledge processing. In addition, the TCP receive packet processing is responsible for placing packets with receive data on the appropriate TCP socket's receive queue or delivering the packet to the first suspended thread waiting for a packet.

TCP Receive Notify

If the application thread needs to process received data from more than one socket, the **nx_tcp_socket_receive_notify** function should be used. This function registers a receive packet callback function for the socket. Whenever a packet is received on the socket, the callback function is executed.

The contents of the callback function are application-specific; however, the function would most likely contain logic to inform the processing thread that a packet is available on the corresponding socket.

TCP Socket Create

TCP sockets are created either during initialization or during runtime by application threads. The initial type of service, time to live, and window size are defined by the ***nx_tcp_socket_create*** service. There are no limits on the number of TCP sockets in an application.

Thread Suspension

As mentioned previously, application threads can suspend while attempting to receive data from a particular TCP port. After a packet is received on that port, it is given to the first thread suspended and that thread is then resumed. An optional timeout is available when suspending on a UDP receive packet, a feature available for most NetX services.

Thread suspension is also available for connection (both client and server), client binding, and disconnection services.

TCP Socket Statistics and Errors

If enabled, the NetX TCP socket software keeps track of several statistics and errors that may be useful to the application. The following statistics and error reports are maintained for each IP/TCP instance:

Total TCP Packets Sent
Total TCP Bytes Sent
Total TCP Packets Received
Total TCP Bytes Received
Total TCP Invalid Packets
Total TCP Receive Packets Dropped
Total TCP Receive Checksum Errors
Total TCP Connections
Total TCP Disconnections
Total TCP Connections Dropped
Total TCP Packet Retransmits
TCP Socket Packets Sent
TCP Socket Bytes Sent
TCP Socket Packets Received
TCP Socket Bytes Received

TCP Socket Packet Retransmits
TCP Socket Packets Queued
TCP Socket Checksum Errors
TCP Socket State
TCP Socket Transmit Queue Depth
TCP Socket Transmit Window Size
TCP Socket Receive Window Size

All these statistics and error reports are available to the application with the ***nx_tcp_info_get*** and ***nx_tcp_socket_info_get*** services.

TCP Socket Control Block NX_TCP_SOCKET

The characteristics of each TCP socket are found in the associated NX_TCP_SOCKET control block, which contains useful information such as the link to the IP data structure, the network connection interface, the bound port, and the receive packet queue. This structure is defined in the ***nx_api.h*** file.

CHAPTER 4

Description of NetX Services

This chapter contains a description of all NetX services in alphabetic order. Service names are designed so all similar services are grouped together. For example, all ARP services are found at the beginning of this chapter.

Note that a BSD-Compatible Socket API is available for legacy application code that cannot take full advantage of the high-performance NetX API. Refer to Appendix D for more information on the BSD-Compatible Socket API.

In the “Return Values” section of each description, values in **BOLD** are not affected by the **NX_DISABLE_ERROR_CHECKING** *define* used to disable the API error checking, while values in non-bold are completely disabled. The “Allowed From” sections indicate from which each NetX service can be called from.

nx_arp_dynamic_entries_invalidate
Invalidate all dynamic entries in the ARP cache 112
nx_arp_dynamic_entry_set
Set dynamic ARP entry 114
nx_arp_enable
Enable Address Resolution Protocol (ARP) 116
nx_arp_gratuitous_send
Send gratuitous ARP request 118
nx_arp_hardware_address_find
Locate physical hardware address given an IP address 120
nx_arp_info_get
Retrieve information about ARP activities 122
nx_arp_ip_address_find
Locate IP address given a physical address 124
nx_arp_static_entries_delete
Delete all static ARP entries 126
nx_arp_static_entry_create
Create static IP to hardware mapping in ARP cache 128

nx_arp_static_entry_delete
Delete static IP to hardware mapping in ARP cache 130
nx_icmp_enable
Enable Internet Control Message Protocol (ICMP) 132
nx_icmp_info_get
Retrieve information about ICMP activities 134
nx_icmp_ping
Send ping request to specified IP address 136
nx_igmp_enable
Enable Internet Group Management Protocol (IGMP) 138
nx_igmp_info_get
Retrieve information about IGMP activities 140
nx_igmp_loopback_disable
Disable IGMP loopback 142
nx_igmp_loopback_enable
Enable IGMP loopback 144
nx_igmp_multicast_interface_join
Join IP interface to specified multicast group 146
nx_igmp_multicast_join
Join IP instance to specified multicast group 148
nx_igmp_multicast_leave
Cause IP instance to leave specified multicast group 150
nx_ip_address_change_notifiy
Notify application if IP address changes 152
nx_ip_address_get
Retrieve IP address and network mask 154
nx_ip_address_set
Set IP address and network mask 156
nx_ip_create
Create an IP instance 158
nx_ip_delete
Delete previously created IP instance 160
nx_ip_driver_direct_command
Issue command to network driver 162
nx_ip_forwarding_disable
Disable IP packet forwarding 164
nx_ip_forwarding_enable
Enable IP packet forwarding 166

nx_ip_fragment_disable
Disable IP packet fragmenting 168
nx_ip_fragment_enable
Enable IP packet fragmenting 170
nx_ip_gateway_address_set
Set Gateway IP address 172
nx_ip_info_get
Retrieve information about IP activities 174
nx_ip_interface_address_get
Retrieve interface IP address 178
nx_ip_interface_address_set
Set interface IP address and network mask 180
nx_ip_interface_attach
Attach network interface to IP instance 182
nx_ip_interface_info_get
Retrieve network interface parameters 184
nx_ip_interface_status_check
Check status of attached IP interface 186
nx_ip_raw_packet_disable
Disable raw packet sending/receiving 188
nx_ip_raw_packet_enable
Enable raw packet sending/receiving 190
nx_ip_raw_packet_interface_send
Send raw IP packet out specified network interface 192
nx_ip_raw_packet_receive
Receive raw IP packet 194
nx_ip_raw_packet_send
Send raw IP packet 196
nx_ip_static_route_add
Add static route 198
nx_ip_static_route_delete
Delete static route 200
nx_ip_status_check
Check status of an IP instance 202
nx_packet_allocate
Allocate packet from specified pool 204
nx_packet_copy
Copy packet 206
nx_packet_data_append
Append data to end of packet 208

nx_packet_data_extract_offset
Extract data from packet via an offset 210
nx_packet_data_retrieve
Retrieve data from packet 212
nx_packet_length_get
Get length of packet data 214
nx_packet_pool_create
Create packet pool in specified memory area 216
nx_packet_pool_delete
Delete previously created packet pool 218
nx_packet_pool_info_get
Retrieve information about a packet pool 220
nx_packet_release
Release previously allocated packet 222
nx_packet_transmit_release
Release a transmitted packet 224
nx_rarp_disable
Disable Reverse Address Resolution Protocol (RARP) 226
nx_rarp_enable
Enable Reverse Address Resolution Protocol (RARP) 228
nx_rarp_info_get
Retrieve information about RARP activities 230
nx_system_initialize
Initialize NetX System 232
nx_tcp_client_socket_bind
Bind client TCP socket to TCP port 234
nx_tcp_client_socket_connect
Connect client TCP socket 236
nx_tcp_client_socket_port_get
Get port number bound to client TCP socket 238
nx_tcp_client_socket_unbind
Unbind TCP client socket from TCP port 240
nx_tcp_enable
Enable TCP component of NetX 242
nx_tcp_free_port_find
Find next available TCP port 244
nx_tcp_info_get
Retrieve information about TCP activities 246
nx_tcp_server_socket_accept
Accept TCP server connection 250

nx_tcp_server_socket_listen
Enable listening for client connection on TCP port 254
nx_tcp_server_socket_relisten
Re-listen for client connection on TCP port 258
nx_tcp_server_socket_unaccept
Unaccept previous server socket connection 262
nx_tcp_server_socket_unlisten
Disable listening for client connection on TCP port 266
nx_tcp_socket_bytes_available
Retrieves number of bytes available for retrieval 270
nx_tcp_socket_create
Create TCP client or server socket 272
nx_tcp_socket_delete
Delete TCP socket 276
nx_tcp_socket_disconnect
Disconnect client and server socket connections 278
nx_tcp_socket_info_get
Retrieve information about TCP socket activities 280
nx_tcp_socket_mss_get
Get MSS of socket 284
nx_tcp_socket_mss_peer_get
Get MSS of socket peer 286
nx_tcp_socket_mss_set
Set MSS of socket 288
nx_tcp_socket_peer_info_get
Retrieve information about peer TCP socket 290
nx_tcp_socket_receive
Receive data from TCP socket 292
nx_tcp_socket_receive_notify
Notify application of received packets 294
nx_tcp_socket_send
Send data through a TCP socket 296
nx_tcp_socket_state_wait
Wait for TCP socket to enter specific state 300
nx_tcp_socket_transmit_configure
Configure socket's transmit parameters 302
nx_tcp_socket_window_update_notify_set
Notify application of window size updates 304
nx_udp_enable
Enable UDP component of NetX 306

nx_udp_free_port_find
Find next available UDP port 308
nx_udp_info_get
Retrieve information about UDP activities 310
nx_udp_packet_info_extract
Extract network parameters from UDP packet 312
nx_udp_socket_bind
Bind UDP socket to UDP port 314
nx_udp_socket_bytes_available
Retrieves number of bytes available for retrieval 316
nx_udp_socket_checksum_disable
Disable checksum for UDP socket 318
nx_udp_socket_checksum_enable
Enable checksum for UDP socket 320
nx_udp_socket_create
Create UDP socket 322
nx_udp_socket_delete
Delete UDP socket 324
nx_udp_socket_info_get
Retrieve information about UDP socket activities 326
nx_udp_socket_interface_send
Send datagram through UDP socket 328
nx_udp_socket_port_get
Pick up port number bound to UDP socket 330
nx_udp_socket_receive
Receive datagram from UDP socket 332
nx_udp_socket_receive_notify
Notify application of each received packet 334
nx_udp_socket_send
Send datagram through UDP socket 336
nx_udp_socket_unbind
Unbind UDP socket from UDP port 338
nx_udp_source_extract
Extract IP and sending port from UDP datagram 340

NETX

nx_arp_dynamic_entries_invalidate

Invalidate all dynamic entries in the ARP cache

Prototype

```
UINT  nx_arp_dynamic_entries_invalidate(NX_IP *ip_ptr);
```

Description

This service invalidates all dynamic ARP entries currently in the ARP cache.

Parameters

ip_ptr	Pointer to previously created IP instance.

Return Values

NX_SUCCESS	(0x00)	Successful ARP cache invalidate.
NX_NOT_ENABLED	(0x14)	ARP is not enabled.
NX_PTR_ERROR	(0x07)	Invalid IP address.
NX_CALLER_ERROR	(0x11)	Caller is not a thread.

Allowed From

Threads

Preemption Possible

No

Example

```
/* Invalidate all dynamic entries in the ARP cache. */
status =  nx_arp_dynamic_entries_invalidate(&ip_0);

/* If status is NX_SUCCESS the dynamic ARP entries were successfully
   invalidated.  */
```

See Also

nx_arp_dynamic_entry_set, nx_arp_enable, nx_arp_gratuitous_send, nx_arp_hardware_address_find, nx_arp_info_get, nx_arp_ip_address_find, nx_arp_static_entries_delete, nx_arp_static_entry_create, nx_arp_static_entry_delete

nx_arp_dynamic_entry_set

Set dynamic ARP entry

Prototype

```
UINT  nx_arp_dynamic_entry_set(NX_IP *ip_ptr, ULONG ip_address,
                               ULONG physical_msw, ULONG physical_lsw);
```

Description

This service allocates a dynamic entry from the ARP cache and sets up the specified IP to physical address mapping. If a zero physical address is specified, an actual ARP request will be broadcast on the network. Also note that this entry will be removed if ARP aging is active or if the ARP cache is exhausted and this is the oldest ARP entry.

If the specified physical address is all zeros, an actual ARP request will be sent for the supplied IP address.

Parameters

ip_ptr	Pointer to previously created IP instance.
ip_address	IP address to map.
physical_msw	Most significant word of the physical address.
physical_lsw	Least significant word of the physical address.

Return Values

NX_SUCCESS	(0x00)	Successful ARP dynamic entry set.
NX_NO_MORE_ENTRIES	(0x17)	No more ARP entries are available in the ARP cache.
NX_PTR_ERROR	(0x07)	Invalid IP instance pointer.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.
NX_IP_ADDRESS_ERROR	(0x21)	Invalid IP address.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads

Preemption Possible

No

Example

```
/* Setup a dynamic ARP entry on the previously created IP Instance
   0. */
status = nx_arp_dynamic_entry_set(&ip_0, IP_ADDRESS(1,2,3,4),
   0x0, 0x1234);

/* If status is NX_SUCCESS, there is now a dynamic mapping between
   the IP address of 1.2.3.4 and the physical hardware address of
   0x0:0x1234. */
```

See Also

nx_arp_dynamic_entries_invalidate, nx_arp_enable, nx_arp_gratuitous_send, nx_arp_hardware_address_find, nx_arp_info_get, nx_arp_ip_address_find, nx_arp_static_entries_delete, nx_arp_static_entry_create, nx_arp_static_entry_delete

nx_arp_enable

Enable Address Resolution Protocol (ARP)

Prototype

```
UINT nx_arp_enable(NX_IP *ip_ptr, VOID *arp_cache_memory,
                   ULONG arp_cache_size);
```

Description

This service initializes the ARP component of NetX for the specific IP instance. ARP initialization includes setting up the ARP cache and various ARP processing routines necessary for sending and receiving ARP requests.

Parameters

ip_ptr	Pointer to previously created IP instance.
arp_cache_memory	Pointer to memory area to place ARP cache.
arp_cache_size	Each ARP entry is approximately 52 bytes, the total number of ARP entries is, therefore, the size divided by 52.

Return Values

NX_SUCCESS	(0x00)	Successful ARP enable.
NX_PTR_ERROR	(0x07)	Invalid IP or cache memory pointer.
NX_SIZE_ERROR	(0x09)	Invalid size of ARP cache memory.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_ALREADY_ENABLED	(0x15)	This component has already been enabled.

Allowed From

Initialization, threads

Preemption Possible

No

Example

```
/* Enable ARP and supply 1024 bytes of ARP cache memory for
   previously created IP Instance 0.  */
status =  nx_arp_enable(&ip_0, (void *) pointer, 1024);

/* If status is NX_SUCCESS, ARP was successfully enabled for this IP
   instance.  */
```

See Also

nx_arp_dynamic_entries_invalidate, nx_arp_dynamic_entry_set, nx_arp_gratuitous_send, nx_arp_hardware_address_find, nx_arp_info_get, nx_arp_ip_address_find, nx_arp_static_entries_delete, nx_arp_static_entry_create, nx_arp_static_entry_delete

nx_arp_gratuitous_send

Send gratuitous ARP request

Prototype

```
UINT  nx_arp_gratuitous_send(NX_IP *ip_ptr,
      VOID (*response_handler)(NX_IP *ip_ptr, NX_PACKET *packet_ptr)));
```

Description

This service sends a gratuitous ARP request. If an ARP response is subsequently received, the supplied response handler is called to process the error.

Parameters

ip_ptr	Pointer to previously created IP instance.
response_handler	Pointer to response handling function. If NX_NULL is supplied, responses are ignored.

Return Values

NX_SUCCESS	(0x00)	Successful gratuitous ARP send.
NX_NO_PACKET	(0x01)	No packet available.
NX_NOT_ENABLED	(0X14)	ARP is not enabled.
NX_IP_ADDRESS_ERROR	(0x14)	Current IP address is invalid.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Caller is not a thread.

Allowed From

Threads

Example

```
/* Send gratuitous ARP without any response handler. */
status =  nx_arp_gratuitous_send(&ip_0, NX_NULL);

/* If status is NX_SUCCESS the gratuitous ARP was successfully
   sent.  */
```

See Also

nx_arp_dynamic_entries_invalidate, nx_arp_dynamic_entry_set, nx_arp_enable, nx_arp_hardware_address_find, nx_arp_info_get, nx_arp_ip_address_find, nx_arp_static_entries_delete, nx_arp_static_entry_create, nx_arp_static_entry_delete

nx_arp_hardware_address_find

Locate physical hardware address given an IP address

Prototype

```
UINT nx_arp_hardware_address_find(NX_IP *ip_ptr,
                        ULONG ip_address, ULONG *physical_msw,
                        ULONG *physical_lsw);
```

Description

This service attempts to find a physical hardware address in the ARP cache that is associated with the supplied IP address.

Parameters

ip_ptr	Pointer to previously created IP instance.
ip_address	IP address to search for.
physical_msw	Pointer to the variable for returning the most significant word of the physical address.
physical_lsw	Pointer to the variable for returning the least significant word of the physical address.

Return Values

NX_SUCCESS	(0x00)	Successful ARP hardware address find.
NX_ENTRY_NOT_FOUND	(0x16)	Mapping was not found in the ARP cache.
NX_IP_ADDRESS_ERROR	(0x21)	Invalid IP address.
NX_PTR_ERROR	(0x07)	Invalid IP or memory pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

No

Example

```
/* Search for the hardware address associated with the IP address of
   1.2.3.4 in the ARP cache of the previously created IP Instance 0.
   */
status = nx_arp_hardware_address_find(&ip_0, IP_ADDRESS(1,2,3,4),
                        &physical_msw, &physical_lsw);

/* If status is NX_SUCCESS, the variables physical_msw and
   physical_lsw contain the hardware address.  */
```

See Also

nx_arp_dynamic_entries_invalidate, nx_arp_dynamic_entry_set, nx_arp_enable, nx_arp_gratuitous_send, nx_arp_info_get, nx_arp_ip_address_find, nx_arp_static_entries_delete, nx_arp_static_entry_create, nx_arp_static_entry_delete

nx_arp_info_get

Retrieve information about ARP activities

Prototype

```
UINT nx_arp_info_get(NX_IP *ip_ptr, ULONG *arp_requests_sent,ULONG
                     *arp_requests_received,ULONG
                     *arp_responses_sent,ULONG
                     *arp_responses_received,ULONG
                     *arp_dynamic_entries,ULONG
                     *arp_static_entries,ULONG
                     *arp_aged_entries,ULONG
                     *arp_invalid_messages);
```

Description

This service retrieves information about ARP activities for the associated IP instance.

If a destination pointer is NX_NULL, that particular information is not returned to the caller.

Parameters

ip_ptr	Pointer to previously created IP instance.
arp_requests_sent	Pointer to destination for the total ARP requests sent from this IP instance.
arp_requests_received	Pointer to destination for the total ARP requests received from the network.
arp_responses_sent	Pointer to destination for the total ARP responses sent from this IP instance.
arp_responses_received	Pointer to the destination for the total ARP responses received from the network.
arp_dynamic_entries	Pointer to the destination for the current number of dynamic ARP entries.
arp_static_entries	Pointer to the destination for the current number of static ARP entries.

arp_aged_entries	Pointer to the destination of the total number of ARP entries that have aged and became invalid.
arp_invalid_messages	Pointer to the destination of the total invalid ARP messages received.

Return Values

NX_SUCCESS	(0x00)	Successful ARP information retrieval.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Initialization, threads, timers

Preemption Possible

No

Example

```
/* Pickup ARP information for ip_0.  */
status =  nx_arp_info_get(&ip_0, &arp_requests_sent,
                          &arp_requests_received,
                          &arp_responses_sent,
                          &arp_responses_received,
                          &arp_dynamic_entries,
                          &arp_static_entries,
                          &arp_aged_entries,
                          &arp_invalid_messages);

/* If status is NX_SUCCESS, the ARP information has been stored in
   the supplied variables.  */
```

See Also

nx_arp_dynamic_entries_invalidate, nx_arp_dynamic_entry_set, nx_arp_enable, nx_arp_gratuitous_send, nx_arp_hardware_address_find, nx_arp_ip_address_find, nx_arp_static_entries_delete, nx_arp_static_entry_create, nx_arp_static_entry_delete

nx_arp_ip_address_find

Locate IP address given a physical address

Prototype

```
UINT nx_arp_ip_address_find(NX_IP *ip_ptr, ULONG *ip_address,
                            ULONG physical_msw, ULONG physical_lsw);
```

Description

This service attempts to find an IP address in the ARP cache that is associated with the supplied physical address.

Parameters

ip_ptr	Pointer to previously created IP instance.
ip_address	Pointer to return IP address, if one is found that has been mapped.
physical_msw	Most significant word of the physical address to search for.
physical_lsw	Least significant word of the physical address to search for.

Return Values

NX_SUCCESS	(0x00)	Successful ARP IP address find
NX_ENTRY_NOT_FOUND	(0x16)	Mapping was not found in the ARP cache.
NX_PTR_ERROR	(0x07)	Invalid IP or memory pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

No

Example

```
/* Search for the IP address associated with the hardware address of
   0x0:0x01234 in the ARP cache of the previously created IP
   Instance 0.  */
status =  nx_arp_ip_address_find(&ip_0, &ip_address,
                                 0x0, 0x1234);

/* If status is NX_SUCCESS, the variables ip_address contains the
   associated IP address.  */
```

See Also

nx_arp_dynamic_entries_invalidate, nx_arp_dynamic_entry_set, nx_arp_enable, nx_arp_gratuitous_send, nx_arp_hardware_address_find, nx_arp_info_get, nx_arp_static_entries_delete, nx_arp_static_entry_create, nx_arp_static_entry_delete

nx_arp_static_entries_delete

Delete all static ARP entries

Prototype

```
UINT nx_arp_static_entries_delete(NX_IP *ip_ptr);
```

Description

This function deletes all static entries in the ARP cache.

Parameters

ip_ptr	Pointer to previously created IP instance.

Return Values

NX_SUCCESS	(0x00)	Static entries are deleted.
NX_PTR_ERROR	(0x07)	Invalid *ip_ptr* pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Initialization, threads

Preemption Possible

No

Example

```
/* Delete all the static ARP entries for IP Instance 0, assuming
   "ip_0" is the NX_IP structure for IP Instance 0.  */
status =  nx_arp_static_entries_delete(&ip_0);

/* If status is NX_SUCCESS all static ARP entries in the ARP cache
   have been
   deleted.  */
```

See Also

nx_arp_dynamic_entries_invalidate, nx_arp_dynamic_entry_set, nx_arp_enable, nx_arp_gratuitous_send, nx_arp_hardware_address_find, nx_arp_info_get, nx_arp_ip_address_find, nx_arp_static_entry_create, nx_arp_static_entry_delete

nx_arp_static_entry_create

Create static IP to hardware mapping in ARP cache

Prototype

```
UINT nx_arp_static_entry_create(NX_IP *ip_ptr, ULONG ip_address,
                          ULONG physical_msw, ULONG physical_lsw);
```

Description

This service creates a static IP-to-physical address mapping in the ARP cache for the specified IP instance. Static ARP entries are not subject to ARP periodic updates.

Parameters

ip_ptr	Pointer to previously created IP instance.
ip_address	IP address to map.
physical_msw	Most significant word of the physical address to map.
physical_lsw	Least significant word of the physical address to map.

Return Values

NX_SUCCESS	(0x00)	Successful ARP static entry create.
NX_NO_MORE_ENTRIES	(0x17)	No more ARP entries are available in the ARP cache.
NX_IP_ADDRESS_ERROR	(0x21)	Invalid IP address.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Initialization, threads

Preemption Possible

No

Example

```
/* Create a static ARP entry on the previously created IP Instance
   0.  */
status =  nx_arp_static_entry_create(&ip_0, IP_ADDRESS(1,2,3,4),
                      0x0, 0x1234);

/* If status is NX_SUCCESS, there is now a static mapping between
   the IP address of 1.2.3.4 and the physical hardware address of
   0x0:0x1234.  */
```

See Also

nx_arp_dynamic_entries_invalidate, nx_arp_dynamic_entry_set, nx_arp_enable, nx_arp_gratuitous_send, nx_arp_hardware_address_find, nx_arp_info_get, nx_arp_ip_address_find, nx_arp_static_entries_delete, nx_arp_static_entry_delete

nx_arp_static_entry_delete

Delete static IP to hardware mapping in ARP cache

Prototype

```
UINT nx_arp_static_entry_delete(NX_IP *ip_ptr, ULONG ip_address,
                                ULONG physical_msw, ULONG physical_lsw);
```

Description

This service finds and deletes a previously created static IP-to-physical address mapping in the ARP cache for the specified IP instance.

Parameters

ip_ptr	Pointer to previously created IP instance.
ip_address	IP address that was mapped statically.
physical_msw	Most significant word of the physical address that was mapped statically.
physical_lsw	Least significant word of the physical address that was mapped statically.

Return Values

NX_SUCCESS	(0x00)	Successful ARP static entry delete.
NX_ENTRY_NOT_FOUND	(0x16)	Static ARP entry was not found in the ARP cache.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

No

Example

```
/* Delete a static ARP entry on the previously created IP Instance
   0.  */
status =  nx_arp_static_entry_delete(&ip_0, IP_ADDRESS(1,2,3,4),
                        0x0, 0x1234);

/* If status is NX_SUCCESS, the previously created static ARP entry
   was successfully deleted.  */
```

See Also

nx_arp_dynamic_entries_invalidate, nx_arp_dynamic_entry_set, nx_arp_enable, nx_arp_gratuitous_send, nx_arp_hardware_address_find, nx_arp_info_get, nx_arp_ip_address_find, nx_arp_static_entries_delete, nx_arp_static_entry_create

nx_icmp_enable

Enable Internet Control Message Protocol (ICMP)

Prototype

```
UINT nx_icmp_enable(NX_IP *ip_ptr);
```

Description

This service enables the ICMP component for the specified IP instance. The ICMP component is responsible for handling Internet error messages and ping requests and replies.

Parameters

ip_ptr	Pointer to previously created IP instance.

Return Values

NX_SUCCESS	(0x00)	Successful ICMP enable.
NX_ALREADY_ENABLED	(0x15)	ICMP is already enabled.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads

Preemption Possible

No

Example

```
/* Enable ICMP on the previously created IP Instance 0.  */
status =  nx_icmp_enable(&ip_0);

/* If status is NX_SUCCESS, ICMP is enabled.  */
```

See Also

nx_icmp_info_get, nx_igmp_loopback_disable, nx_igmp_loopback_enable, nx_icmp_ping

nx_icmp_info_get

Retrieve information about ICMP activities

Prototype

```
UINT nx_icmp_info_get(NX_IP *ip_ptr,
                      ULONG *pings_sent,
                      ULONG *ping_timeouts,
                      ULONG *ping_threads_suspended,
                      ULONG *ping_responses_received,
                      ULONG *icmp_checksum_errors,
                      ULONG *icmp_unhandled_messages);
```

Description

This service retrieves information about ICMP activities for the specified IP instance.

If a destination pointer is NX_NULL, that particular information is not returned to the caller.

Parameters

ip_ptr	Pointer to previously created IP instance.
pings_sent	Pointer to destination for the total number of pings sent.
ping_timeouts	Pointer to destination for the total number of ping timeouts.
ping_threads_suspended	Pointer to destination of the total number of threads suspended on ping requests.
ping_responses_received	Pointer to destination of the total number of ping responses received.
icmp_checksum_errors	Pointer to destination of the total number of ICMP checksum errors.
icmp_unhandled_messages	Pointer to destination of the total number of un-handled ICMP messages.

Return Values

NX_SUCCESS	(0x00)	Successful ICMP information retrieval.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Initialization, threads, and timers

Preemption Possible

No

Example

```
/* Retrieve ICMP information from previously created IP Instance 0.
   */
status =  nx_icmp_info_get(&ip_0, &pings_sent, &ping_timeouts,
   &ping_threads_suspended, &ping_responses_received,
   &icmp_checksum_errors, &icmp_unhandled_messages);

/* If status is NX_SUCCESS, ICMP information was retrieved.  */
```

See Also

nx_icmp_enable, nx_igmp_loopback_disable, nx_igmp_loopback_enable, nx_icmp_ping

nx_icmp_ping

Send ping request to specified IP address

Prototype

```
UINT nx_icmp_ping(NX_IP *ip_ptr, ULONG ip_address,
                  CHAR *data, ULONG data_size,
                  NX_PACKET **response_ptr, ULONG wait_option);
```

Description

This service sends a ping request to the specified IP address and waits for the specified amount of time for a ping response message. If no response is received, an error is returned. Otherwise, the entire response message, including the ICMP header, is returned in the variable pointed to by response_ptr.

If NX_SUCCESS is returned, the application is responsible for releasing the received packet after it is no longer needed.

Parameters

ip_ptr	Pointer to previously created IP instance.
ip_address	IP address to ping.
data	Pointer to data area for ping message.
data_size	Number of bytes in the ping data
response_ptr	Pointer to packet pointer to return the ping response message in.
wait_option	Defines how long to wait for a ping response. Legal values are: 1 through 0xFFFFFFFE.

Return Values

NX_SUCCESS	(0x00)	Successful ping. Response message pointer was placed in the variable pointed to by response_ptr.
NX_NO_PACKET	(0x01)	Unable to allocate a ping request packet.

NX_OVERFLOW	(0x03)	Specified data area exceeds the default packet size for this IP instance.
NX_NO_RESPONSE	(0x29)	Requested IP did not respond.
NX_WAIT_ABORTED	(0x1A)	Requested suspension was aborted by a call to *tx_thread_wait_abort*.
NX_IP_ADDRESS_ERROR	(0x21)	Invalid IP address.
NX_PTR_ERROR	(0x07)	Invalid IP or response pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

Yes

Example

```
/* Issue a ping to IP address 1.2.3.5 from the previously created IP
   Instance 0.  */
status =  nx_icmp_ping(&ip_0, IP_ADDRESS(1,2,3,5), "abcd", 4,
                       &response_ptr, 10);

/* If status is NX_SUCCESS, a ping response was received from IP
   address 1.2.3.5 and the response packet is contained in the
   packet pointed to by response_ptr. It should have the same "abcd"
   four bytes of data.  */
```

See Also

nx_icmp_enable, nx_icmp_info_get

nx_igmp_enable

Enable Internet Group Management Protocol (IGMP)

Prototype

```
UINT nx_igmp_enable(NX_IP *ip_ptr);
```

Description

This service enables the IGMP component on the specified IP instance. The IGMP component is responsible for providing support for IP multicast operations.

Parameters

ip_ptr	Pointer to previously created IP instance.

Return Values

NX_SUCCESS	(0x00)	Successful IGMP enable.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_ALREADY_ENABLED	(0x15)	This component has already been enabled.

Allowed From

Initialization, threads

Preemption Possible

No

Example

```
/* Enable IGMP on the previously created IP Instance 0.  */
status =  nx_igmp_enable(&ip_0);

/* If status is NX_SUCCESS, IGMP is enabled.  */
```

See Also

nx_igmp_info_get,nx_igmp_loopback_disable, nx_igmp_loopback_enable, nx_igmp_multicast_interface_join, nx_igmp_multicast_join, nx_igmp_multicast_leave

nx_igmp_info_get

Retrieve information about IGMP activities

Prototype

```
UINT nx_igmp_info_get(NX_IP *ip_ptr,
                      ULONG *igmp_reports_sent,
                      ULONG *igmp_queries_received,
                      ULONG *igmp_checksum_errors,
                      ULONG *current_groups_joined);
```

Description

This service retrieves information about IGMP activities for the specified IP instance.

If a destination pointer is NX_NULL, that particular information is not returned to the caller.

Parameters

ip_ptr	Pointer to previously created IP instance.
igmp_reports_sent	Pointer to destination for the total number of ICMP reports sent.
igmp_queries_received	Pointer to destination for the total number of queries received by multicast router.
igmp_checksum_errors	Pointer to destination of the total number of IGMP checksum errors on receive packets.
current_groups_joined	Pointer to destination of the current number of groups joined through this IP instance.

Return Values

NX_SUCCESS	(0x00)	Successful IGMP information retrieval.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Initialization, threads, and timers

Preemption Possible

No

Example

```
/* Retrieve IGMP information from previously created IP Instance 0.  */
status =  nx_igmp_info_get(&ip_0, &igmp_reports_sent,
                      &igmp_queries_received,
                      &igmp_checksum_errors, &current_groups_joined);

/* If status is NX_SUCCESS, IGMP information was retrieved.  */
```

See Also

nx_igmp_enable, nx_igmp_loopback_disable, nx_igmp_loopback_enable, nx_igmp_multicast_interface_join, nx_igmp_multicast_join, nx_igmp_multicast_leave

nx_igmp_loopback_disable

Disable IGMP loopback

Prototype

```
UINT  nx_igmp_loopback_disable(NX_IP *ip_ptr);
```

Description

This service disables IGMP loopback for all subsequent multicast groups joined.

Parameters

ip_ptr	Pointer to previously created IP instance.

Return Values

NX_SUCCESS	(0x00)	Successful IGMP loopback disable.
NX_NOT_ENABLED	(0x14)	IGMP is not enabled.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Caller is not a thread or initialization.

Allowed From

Initialization, threads

Example

```
/* Disable IGMP loopback for all subsequent multicast groups joined. */
status =  nx_igmp_loopback_disable(&ip_0);

/* If status is NX_SUCCESS IGMP loopback is disabled.  */
```

See Also

nx_igmp_enable, nx_igmp_info_get, nx_igmp_loopback_enable, nx_igmp_multicast_interface_join, nx_igmp_multicast_join, nx_igmp_multicast_leave

nx_igmp_loopback_enable

Enable IGMP loopback

Prototype

```
UINT  nx_igmp_loopback_enable(NX_IP *ip_ptr);
```

Description

This service enables IGMP loopback for all subsequent multicast groups joined.

Parameters

ip_ptr	Pointer to previously created IP instance.

Return Values

NX_SUCCESS	(0x00)	Successful IGMP loopback disable.
NX_NOT_ENABLED	(0x14)	IGMP is not enabled.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Caller is not a thread or initialization.

Allowed From

Initialization, threads

Example

```
/* Enable IGMP loopback for all subsequent multicast
   groups joined. */
status =  nx_igmp_loopback_enable(&ip_0);

/* If status is NX_SUCCESS IGMP loopback is enabled.  */
```

See Also

nx_igmp_enable, nx_igmp_info_get, nx_igmp_loopback_disable, nx_igmp_multicast_interface_join, nx_igmp_multicast_join, nx_igmp_multicast_leave

nx_igmp_multicast_interface_join

Join IP interface to specified multicast group

Prototype

```
UINT nx_igmp_multicast_interface_join(NX_IP *ip_ptr, ULONG group_address,
                                      UINT interface_index)
```

Description

This service joins an IP instance to the specified multicast group via a specified network interface. An internal counter is maintained to keep track of the number of times the same group has been joined. After joined, the IGMP component will allow reception of IP packets with this group address via the specified network interface and also report to routers that this IP is a member of this multicast group. The IGMP membership join, report, and leave messages are also sent via the specified network interface.

Parameters

ip_ptr	Pointer to previously created IP instance.
group_address	Class D IP multicast group address to join in host byte order.
interface_index	Interface attached to NetX instance.

Return Values

NX_SUCCESS	(0x00)	Successful multicast group join.
NX_NO_MORE_ENTRIES	(0x17)	No more multicast groups can be joined, maximum exceeded.
NX_INVALID_INTERFACE	(0X4c0	Interface index points to an invalid network interface.
NX_IP_ADDRESS_ERROR	(0x21)	Multicast group address provided is not a valid class D address.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	IP multicast support is not enabled.

Allowed From

Threads

Preemption Possible

No

Example

```
/* Previously created IP Instance joins the multicast group 244.0.0.200, via
   the interface at index 1 in the IP task interface list.   */
status = nx_igmp_multicast_join_interface(&ip, IP_ADDRESS(244,0,0,200), 1);

/* If status is NX_SUCCESS, the IP instance has successfully joined the
   multicast group. */
```

See Also

nx_igmp_enable, nx_igmp_info_get, nx_igmp_loopback_disable, nx_igmp_loopback_enable, nx_igmp_multicast_join, nx_igmp_multicast_leave

nx_igmp_multicast_join

Join IP instance to specified multicast group

Prototype

```
UINT nx_igmp_multicast_join(NX_IP *ip_ptr, ULONG group_address);
```

Description

This service joins an IP instance to the specified multicast group. An internal counter is maintained to keep track of the number of times the same group has been joined. After joined, the IGMP component will allow reception of IP packets with this group address and report to routers that this IP is a member of this multicast group.

Parameters

ip_ptr	Pointer to previously created IP instance.
group_address	Class D IP multicast group address to join.

Return Values

NX_SUCCESS	(0x00)	Successful multicast group join.
NX_NO_MORE_ENTRIES	(0x17)	No more multicast groups can be joined, maximum exceeded.
NX_IP_ADDRESS_ERROR	(0x21)	Invalid IP group address.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Initialization, threads

Preemption Possible

No

Example

```
/* Previously created IP Instance 0 joins the multicast group
   224.0.0.200. */
status =  nx_igmp_multicast_join(&ip_0, IP_ADDRESS(224,0,0,200);

/* If status is NX_SUCCESS, this IP instance has successfully joined
   the multicast group 224.0.0.200.  */
```

See Also

nx_igmp_enable, nx_igmp_info_get, nx_igmp_loopback_disable, nx_igmp_loopback_enable, nx_igmp_multicast_interface_join, nx_igmp_multicast_leave

nx_igmp_multicast_leave

Cause IP instance to leave specified multicast group

Prototype

```
UINT nx_igmp_multicast_leave(NX_IP *ip_ptr,
                             ULONG group_address);
```

Description

This service causes an IP instance to leave the specified multicast group, if the number of leave requests matches the number of join requests. Otherwise, the internal join count is simply decremented.

Parameters

ip_ptr	Pointer to previously created IP instance.
group_address	Multicast group to leave.

Return Values

NX_SUCCESS	(0x00)	Successful multicast group join.
NX_ENTRY_NOT_FOUND	(0x16)	Previous join request was not found.
NX_IP_ADDRESS_ERROR	(0x21)	Invalid IP group address.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

No

Example

```
/* Cause IP instance to leave the multicast group 224.0.0.200.  */
status =  nx_igmp_multicast_leave(&ip_0, IP_ADDRESS(224,0,0,200);

/* If status is NX_SUCCESS, this IP instance has successfully left
   the multicast group 224.0.0.200.  */
```

See Also

nx_igmp_enable, nx_igmp_info_get, nx_igmp_loopback_disable, nx_igmp_loopback_enable, nx_igmp_multicast_interface_join,, nx_igmp_multicast_join

nx_ip_address_change_notifiy

Notify application if IP address changes

Prototype

```
UINT  nx_ip_address_change_notify(NX_IP *ip_ptr,
            VOID (*change_notify)(NX_IP *, VOID *),
            VOID *additional_info);
```

Description

This service registers an application notification function that is called whenever the IP address is changed.

Parameters

ip_ptr	Pointer to previously created IP instance.
change_notify	Pointer to IP change notification function. If this parameter is NX_NULL, IP address change notification is disabled.
additional_info	Pointer to optional additional information that is also supplied to the notification function when the IPaddress is changed.

Return Values

NX_SUCCESS	(0x00)	Successful IP address change notification.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.

Allowed From

Initialization, threads, timers

Example

```
/* Register the function "my_ip_changed" to be called whenever the
   IP address is changed. */
status =  nx_ip_address_change_notify(&ip_0, my_ip_changed,
                                      NX_NULL);

/* If status is NX_SUCCESS, the "my_ip_changed" function will be
   called whenever the IP address changes.  */
```

See Also

nx_ip_address_get, nx_ip_address_set, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_info_get, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check

nx_ip_address_get

Retrieve IP address and network mask

Prototype

```
UINT nx_ip_address_get(NX_IP *ip_ptr, ULONG *ip_address,
                       ULONG *network_mask);
```

Description

This service retrieves the IP and network mask for the specified IP instance.

Parameters

ip_ptr	Pointer to previously created IP instance.
ip_address	Pointer to destination for IP address.
network_mask	Pointer to destination for network mask.

Return Values

NX_SUCCESS	(0x00)	Successful IP address get.
NX_PTR_ERROR	(0x07)	Invalid IP or return variable pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads, timers

Preemption Possible

No

Example

```
/* Get the IP address and network mask from the previously created
   IP instance 0.  */
status =  nx_ip_address_get(&ip_0, &ip_address, &network_mask);

/* If status is NX_SUCCESS, the variables ip_address and
   network_mask contain the IP and network mask respectively.  */
```

See Also

nx_ip_address_change_notify, nx_ip_address_set, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_info_get, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check

nx_ip_address_set

Set IP address and network mask

Prototype

```
UINT nx_ip_address_set(NX_IP *ip_ptr, ULONG ip_address,
                       ULONG network_mask);
```

Description

This service sets the IP address and network mask for the specified IP instance.

Parameters

ip_ptr	Pointer to previously created IP instance.
ip_address	New IP address.
network_mask	New network mask.

Return Values

NX_SUCCESS	(0x00)	Successful IP address set.
NX_IP_ADDRESS_ERROR	(0x21)	Invalid IP address.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads

Preemption Possible

No

Example

```
/* Set the IP address and network mask to 1.2.3.4 and 0xFF for the
   previously created IP instance 0.  */
status =  nx_ip_address_set(&ip_0, IP_ADDRESS(1,2,3,4),
                            0xFFFFFF00UL);

/* If status is NX_SUCCESS, the IP instance now has an IP address of
   1.2.3.4 and a network mask of 0xFF.   */
```

See Also

nx_ip_address_change_notify, nx_ip_address_get, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_interface_address_get, nx_ip_info_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_receive, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check

nx_ip_create

Create an IP instance

Prototype

```
UINT nx_ip_create(NX_IP *ip_ptr, CHAR *name, ULONG ip_address,
                  ULONG network_mask, NX_PACKET_POOL *default_pool,
                  VOID (*ip_network_driver)(NX_IP_DRIVER *),
                  VOID *memory_ptr, ULONG memory_size,
                  UINT priority);
```

Description

This service creates an IP instance with the user supplied IP address and network driver. In addition, the application must supply a previously created packet pool for the IP instance to use for internal packet allocation. Note that the supplied application network driver is not called until this IP's thread executes.

Parameters

ip_ptr	Pointer to control block to create a new IP instance.
name	Name of this new IP instance.
ip_address	IP address for this new IP instance.
network_mask	Mask to delineate the network portion of the IP address for sub-netting and super-netting uses.
default_pool	Pointer to control block of previously created NetX packet pool.
ip_network_driver	User-supplied network driver used to send and receive IP packets.
memory_ptr	Pointer to memory area for the IP helper thread's stack area.
memory_size	Number of bytes in the memory area for the IP helper thread's stack.
priority	Priority of IP helper thread.

Return Values

NX_SUCCESS	(0x00)	Successful IP instance creation.
NX_IP_INTERNAL_ERROR	(0x20)	An internal IP system resource was not able to be created

		causing the IP create service to fail.
NX_PTR_ERROR	(0x07)	Invalid IP, network driver address, packet pool, or memory pointer.
NX_SIZE_ERROR	(0x09)	The supplied stack size is too small.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_IP_ADDRESS_ERROR	(0x21)	The supplied IP address is invalid.

Allowed From

Initialization, threads

Preemption Possible

Yes

Example

```
/* Create an IP instance with an IP address of 1.2.3.4 and a network
   mask of 0xFFFFFF00UL. The "ethernet_driver" specifies the entry
   point of the application specific network driver and the
   "stack_memory_ptr" specifies the start of a 1024 byte memory area
   that is used for this IP instance's helper thread.  */
status = nx_ip_create(&ip_0, "NetX IP Instance 0",
                      IP_ADDRESS(1, 2, 3, 4),
                      0xFFFFFF00UL, &pool_0, ethernet_driver,
                      stack_memory_ptr, 1024, 1);

/* If status is NX_SUCCESS, the IP instance has been created.  */
```

See Also

nx_ip_address_change_notify, nx_ip_address_get, nx_ip_address_set, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_raw_packet_interface_send, nx_ip_interface_status_check, nx_ip_info_get, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check

nx_ip_delete

Delete previously created IP instance

Prototype

```
UINT nx_ip_delete(NX_IP *ip_ptr);
```

Description

This service deletes a previously created IP instance and in the process releases all of the system resources owned by the IP instance.

Parameters

ip_ptr	Pointer to previously created IP instance.

Return Values

NX_SUCCESS	(0x00)	Successful IP deletion.
NX_SOCKETS_BOUND	(0x28)	This IP instance still has UDP or TCP sockets bound to it. All sockets must be unbound and deleted prior to deleting the IP instance.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Threads

Preemption Possible

Yes

Example

```
/* Delete a previously created IP instance. */
status = nx_ip_delete(&ip_0);

/* If status is NX_SUCCESS, the IP instance has been deleted.  */
```

See Also

nx_ip_address_change_notify, nx_ip_address_get, nx_ip_address_set, nx_ip_create, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_info_get, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check

nx_ip_driver_direct_command

Issue command to network driver

Prototype

```
UINT nx_ip_driver_direct_command(NX_IP *ip_ptr, UINT command,
                          ULONG *return_value_ptr);
```

Description

This service provides a direct interface to the application's network driver specified during the ***nx_ip_create*** call. Application-specific commands can be used providing their numeric value is greater than or equal to NX_LINK_USER_COMMAND.

Parameters

ip_ptr	Pointer to previously created IP instance.
command	Numeric command code. Standard commands are defined as follows:

NX_LINK_GET_STATUS	(10)
NX_LINK_GET_SPEED	(11)
NX_LINK_GET_DUPLEX_TYPE	(12)
NX_LINK_GET_ERROR_COUNT	(13)
NX_LINK_GET_RX_COUNT	(14)
NX_LINK_GET_TX_COUNT	(15)
NX_LINK_GET_ALLOC_ERRORS	(16)

return_value_ptr	Pointer to return variable in the caller.

Return Values

NX_SUCCESS	(0x00)	Successful network driver direct command.
NX_UNHANDLED_COMMAND	(0x44)	Unhandled or unimplemented network driver command.
NX_PTR_ERROR	(0x07)	Invalid IP or return value pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads, timers

Preemption Possible

No

Example

```
/* Make a direct call to the application-specific network driver for
   the previously created IP instance. For this example, the network
   driver is interrogated for the link status.  */
status = nx_ip_driver_direct_command(&ip_0, NX_LINK_GET_STATUS,
                    &link_status);

/* If status is NX_SUCCESS, the link_status variable contains a
   NX_TRUE or NX_FALSE value representing the status of the physical
   link.  */
```

See Also

nx_ip_address_change_notify, nx_ip_address_get, nx_ip_address_set, nx_ip_create, nx_ip_delete, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_info_get, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check

nx_ip_forwarding_disable

Disable IP packet forwarding

Prototype

```
UINT nx_ip_forwarding_disable(NX_IP *ip_ptr);
```

Description

This service disables forwarding IP packets inside the NetX IP component.

Parameters

ip_ptr	Pointer to previously created IP instance.

Return Values

NX_SUCCESS	(0x00)	Successful IP forwarding disable.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads, timers

Preemption Possible

No

Example

```
/* Disable IP forwarding on this IP instance.  */
status = nx_ip_forwarding_disable(&ip_0);

/* If status is NX_SUCCESS, IP forwarding has been disabled on the
   previously created IP instance.  */
```

See Also

nx_ip_address_change_notify, nx_ip_address_get, nx_ip_address_set, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_info_get, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check

nx_ip_forwarding_enable

Enable IP packet forwarding

Prototype

```
UINT nx_ip_forwarding_enable(NX_IP *ip_ptr);
```

Description

This service enables forwarding IP packets inside the NetX IP component.

Parameters

ip_ptr	Pointer to previously created IP instance.

Return Values

NX_SUCCESS	(0x00)	Successful IP forwarding enable.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads, timers

Preemption Possible

No

Example

```
/* Enable IP forwarding on this IP instance.  */
status = nx_ip_forwarding_enable(&ip_0);

/* If status is NX_SUCCESS, IP forwarding has been enabled on the
   previously created IP instance.  */
```

See Also

nx_ip_address_change_notify, nx_ip_address_get, nx_ip_address_set, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_info_get, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check

nx_ip_fragment_disable

Disable IP packet fragmenting

Prototype

```
UINT nx_ip_fragment_disable(NX_IP *ip_ptr);
```

Description

This service disables IP packet fragmenting and un-fragmenting.

Parameters

ip_ptr	Pointer to previously created IP instance.

Return Values

NX_SUCCESS	(0x00)	Successful IP fragment disable.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads, timers

Preemption Possible

No

Example

```
/* Disable IP fragmenting on this IP instance.  */
status = nx_ip_fragment_disable(&ip_0);

/* If status is NX_SUCCESS, disables IP fragmenting on the
   previously created IP instance.  */
```

See Also

nx_ip_address_change_notify, nx_ip_address_get, nx_ip_address_set, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_info_get, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check

nx_ip_fragment_enable

Enable IP packet fragmenting

Prototype

```
UINT nx_ip_fragment_enable(NX_IP *ip_ptr);
```

Description

This service enables IP packet fragmenting and un-fragmenting.

Parameters

ip_ptr	Pointer to previously created IP instance.

Return Values

NX_SUCCESS	(0x00)	Successful IP fragment enable.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads, timers

Preemption Possible

No

Example

```
/* Enable IP fragmenting on this IP instance.  */
status = nx_ip_fragment_enable(&ip_0);

/* If status is NX_SUCCESS, IP fragmenting has been enabled on the
   previously created IP instance.  */
```

See Also

nx_ip_address_change_notify, nx_ip_address_get, nx_ip_address_set, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_gateway_address_set, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_info_get, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check

nx_ip_gateway_address_set

Set Gateway IP address

Prototype

```
UINT  nx_ip_gateway_address_set(NX_IP *ip_ptr, ULONG ip_address);
```

Description

This service sets the Gateway IP address to the specified IP address. All out-of-network IP addresses are routed to this IP address for transmission.

Parameters

ip_ptr	Pointer to previously created IP instance.
ip_address	IP address of the Gateway.

Return Values

NX_SUCCESS	(0x00)	Successful Gateway IP address set.
NX_PTR_ERROR	(0x07)	Invalid IP instance pointer.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.
NX_IP_ADDRESS_ERROR	(0x21)	Invalid IP address.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Threads

Preemption Possible

No

Example

```
/* Setup the Gateway address for previously created IP
   Instance 0. */
status = nx_ip_gateway_address_set(&ip_0, IP_ADDRESS(1,2,3,99);

/* If status is NX_SUCCESS, all out-of-network send requests are
   routed to 1.2.3.99. */
```

See Also

nx_ip_address_change_notify, nx_ip_address_get, nx_ip_address_set, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_info_get, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check

nx_ip_info_get

Retrieve information about IP activities

Prototype

```
UINT nx_ip_info_get(NX_IP *ip_ptr,
                    ULONG *ip_total_packets_sent,
                    ULONG *ip_total_bytes_sent,
                    ULONG *ip_total_packets_received,
                    ULONG *ip_total_bytes_received,
                    ULONG *ip_invalid_packets,
                    ULONG *ip_receive_packets_dropped,
                    ULONG *ip_receive_checksum_errors,
                    ULONG *ip_send_packets_dropped,
                    ULONG *ip_total_fragments_sent,
                    ULONG *ip_total_fragments_received);
```

Description

This service retrieves information about IP activities for the specified IP instance.

If a destination pointer is NX_NULL, that particular information is not returned to the caller.

Parameters

ip_ptr	Pointer to previously created IP instance.
ip_total_packets_sent	Pointer to destination for the total number of IP packets sent.
ip_total_bytes_sent	Pointer to destination for the total number of bytes sent.
ip_total_packets_received	Pointer to destination of the total number of IP receive packets.
ip_total_bytes_received	Pointer to destination of the total number of IP bytes received.
ip_invalid_packets	Pointer to destination of the total number of invalid IP packets.
ip_receive_packets_dropped	Pointer to destination of the total number of receive packets dropped.
ip_receive_checksum_errors	Pointer to destination of the total number of checksum errors in receive packets.
ip_send_packets_dropped	Pointer to destination of the total number of send packets dropped.

ip_total_fragments_sent	Pointer to destination of the total number of fragments sent.
ip_total_fragments_received	Pointer to destination of the total number of fragments received.

Return Values

NX_SUCCESS	(0x00)	Successful IP information retrieval.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.

Allowed From

Initialization, threads, and timers

Preemption Possible

No

Example

```
/* Retrieve IP information from previously created IP Instance 0.
   */
status =  nx_ip_info_get(&ip_0,
                         &ip_total_packets_sent,
                         &ip_total_bytes_sent,
                         &ip_total_packets_received,
                         &ip_total_bytes_received,
                         &ip_invalid_packets,
                         &ip_receive_packets_dropped,
                         &ip_receive_checksum_errors,
                         &ip_send_packets_dropped,
                         &ip_total_fragments_sent,
                         &ip_total_fragments_received);

/* If status is NX_SUCCESS, IP information was retrieved.  */
```

See Also

nx_ip_address_change_notify, nx_ip_address_get, nx_ip_address_set, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_receive,

nx_ip_raw_packet_send, nx_ip_static_route_add,
nx_ip_static_route_delete, nx_ip_status_check

NETX

nx_ip_interface_address_get

Retrieve interface IP address

Prototype

```
UINT nx_ip_interface_address_get NX_IP *ip_ptr, ULONG interface_id,
                                 ULONG *ip_address, ULONG *network_mask)
```

Description

This service retrieves the IP address of a specified network interface.

The specified interface, if not the primary interface, must be previously attached to the IP instance.

Parameters

ip_ptr	Pointer to previously created IP instance.
interface_id	Interface index attached to NetX instance.
ip_address	Pointer to destination for the device interface IP address.
network_mask	Pointer to destination for the device interface network mask.

Return Values

NX_SUCCESS	(0x00)	Successful IP address get.
NX_INVALID_INTERFACE	**(0X4c)**	Specified network interface is invalid.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.

Allowed From

Initialization, threads, timers

Preemption Possible

No

Example

```
/* Get device IP address and network mask for the specified interface
   (index 1 in IP instance list of interfaces).  */
status =  nx_ip_interface_address_get(ip_ptr,1, &ip_address,
   &network_mask);

/* If status is NX_SUCCESS the interface address was successfully
   retrieved.  */
```

See Also

nx_ip_address_change_notify, nx_ip_ address_get, nx_ip_ address_set, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_info_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check

nx_ip_interface_address_set

Set interface IP address and network mask

Prototype

```
UINT nx_ip_interface_address_set NX_IP *ip_ptr, ULONG interface_id,
                                 ULONG ip_address, ULONG network_mask)
```

Description

This service sets the IP address and network mask for the specified IP interface.

The specified interface must be previously attached to the IP instance.

Parameters

ip_ptr	Pointer to previously created IP instance.
interface_id	Interface index attached to NetX instance.
ip_address	New network interface IP address.
network_mask	New interface network mask.

Return Values

NX_SUCCESS	(0x00)	Successful IP address set.
NX_INVALID_INTERFACE	**(0X4C)**	Specified network interface is invalid.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_PTR_ERROR	(0x07)	Invalid pointers.

Allowed From

Initialization, threads

Preemption Possible

No

Example

```
/* Set device IP address and network mask for the specified interface
   (index 1 in IP instance list of interfaces).  */
status =  nx_ip_interface_address_set(ip_ptr,1, ip_address, network_mask);

/* If status is NX_SUCCESS the interface IP address and mask was
   successfully set.  */
```

See Also

nx_ip_address_change_notify, nx_ip_ address_get, nx_ip_ address_set, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_info_get, nx_ip_interface_address_get, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check

nx_ip_interface_attach

Attach network interface to IP instance

Prototype

```
UINT nx_ip_interface_attach(NX_IP *ip_ptr, CHAR *interface_name,
                            ULONG ip_address, ULONG network_mask,
                            VOID(*ip_link_driver)(struct
                            NX_IP_DRIVER_STRUCT *));
```

Description

This function adds a physical network interface to the IP interface table. Note the IP task is created with the primary interface so each additional interface is secondary to the primary interface. NX_MAX_PHYSICAL_INTERFACES must be set to the number of interfaces. The default is 1 for the primary interface.

ip_ptr must point to a valid NetX IP structure. NX_MAX_PHYSICAL_INTERFACES must be configured for the number of network interfaces for the IP instance. The default value is one.

Parameters

ip_ptr	Pointer to previously created IP instance.
interface_name	Pointer to device name buffer.
ip_address	Device IP address in host byte order.
network_mask	Device network mask in host byte order.
ip_link_driver	Ethernet driver for the interface.

Return Values

NX_SUCCESS	(0x00)	Entry is added to static routing table.
NX_NO_MORE_ENTRIES	**(0X17)**	Max number of interfaces. NX_MAX_PHYSICAL_INTERFACES is exceeded.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_PTR_ERROR	(0x07)	Invalid pointer input.
NX_IP_ADDRESS_ERROR	(0x21)	Invalid IP address input.

Allowed From

Initialization, threads

Preemption Possible

No

Example

```
/* Attach secondary interface for device IP address 192.168.1.68 with the
   specified ethernet driver.  */
status =  nx_ip_interface_attach(ip_ptr, "secondary_port",
                                 IP_ADDRESS(192,168,1,68),
                                 0xFFFFFF00UL, nx_etherDriver_mcf5485);

/* If status is NX_SUCCESS the interface was successfully added to the IP
   task interface  table.  */
```

See Also

nx_ip_address_change_notify, nx_ip_ address_get, nx_ip_ address_set, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_info_get, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check

nx_ip_interface_info_get

Retrieve network interface parameters

Prototype

```
UINT nx_ip_interface_info_get(NX_IP *ip_ptr, UINT interface_index, CHAR
                              **interface_name,
                              ULONG *ip_address, ULONG *network_mask,
                              ULONG *mtu_size,
                              ULONG *physical_address_msw, ULONG
                              *physical_address_lsw);
```

Description

This function retrieves information on network parameters for the specified interface. All network data is retrieved in host byte order.

ip_ptr must point to a valid NetX IP structure. The specified interface, if not the primary interface, must be previously attached to the IP instance.

Parameters

ip_ptr	Pointer to previously created IP instance.
interface_interface	Index specifying network interface.
interface_name	Pointer to destination for interface name.
ip_address	Pointer to destination for network interface IP address.
network_mask	Pointer to destination for network interface mask.
mtu_size	Pointer to destination for maximum transfer unit for the IP task. Differs from the driver MTU by the additional size for the Ethernet header.
physical_address_msw	Pointer to destination for MSB of interface MAC address.
physical_address_lsw	Pointer to destination for LSB of interface MAC address.

Return Values

NX_SUCCESS	(0x00)	Entry is added to static routing table.

NX_PTR_ERROR	(0x07)	Invalid pointer input.
NX_INVALID_INTERFACE	(0x4C)	Invalid IP pointer.

Allowed From

Initialization, threads, timers, ISRs

Preemption Possible

No

Example

```
/* Retrieve interface parameters for the specified interface (index 1 in IP
   instance list of interfaces).  */
status =  nx_ip_interface_info_get(ip_ptr, 1, &name_ptr, &ip_address,
                                   &network_mask,
                                   &mtu_size, &physical_address_msw,
                                   &physical_address_lsw);

/* If status is NX_SUCCESS the interface was successfully added to the IP
   task interface table.  */
```

See Also

nx_ip_address_change_notify, nx_ip_ address_get, nx_ip_ address_set, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_info_get, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_status_check, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check

nx_ip_interface_status_check

Check status of attached IP interface

Prototype

```
UINT nx_ip_interface_status_check NX_IP *ip_ptr, UINT interface_index,
                          ULONG needed_status,
                          ULONG *actual_status,
                          ULONG wait_option);
```

Description

This service checks and optionally waits for the specified status of the interface corresponding to the interface index attached to the IP instance. Note: the *nx_ip_status_check* service can provide the same information but defaults to the primary interface on the IP instance.

ip_ptr must point to a valid NetX IP structure. The specified interface, if not the primary interface, must be previously attached to the IP instance.

Parameters

ip_ptr	Pointer to previously created IP instance.
interface_index	Index specifying network interface.
needed_status	IP status requested, defined in bit-map form as follows: NX_IP_INITIALIZE_DONE (0x0001) NX_IP_ADDRESS_RESOLVED (0x0002) NX_IP_LINK_ENABLED (0x0004) NX_IP_ARP_ENABLED (0x0008) NX_IP_UDP_ENABLED (0x0010) NX_IP_TCP_ENABLED (0x0020) NX_IP_IGMP_ENABLED (0x0040) NX_IP_RARP_COMPLETE (0x0080)
actual_status	Pointer to the actual bits set.
wait_option	Defines how the service behaves if the requested status bits are not available. The wait options are defined as follows: NX_NO_WAIT 0x00000000) timeout value (0x00000001 through 0xFFFFFFFE)

Return Values

NX_SUCCESS	(0x00)	Successful IP status check.
NX_PTR_ERROR	(0x07)	IP pointer is or has become invalid or actual status pointer is invalid.
NX_NOT_SUCCESSFUL	(0x43)	Status request was not satisfied within the timeout specified.
NX_WAIT_ABORTED	(0x1A)	Requested suspension was aborted by a call to *tx_thread_wait_abort*.
NX_INVALID_INTERFACE	(0x4C)	Invalid interface.
NX_OPTION_ERROR	(0x0a)	Invalid needed status option.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads

Preemption Possible

No

Example

```
/* Wait 10 ticks for the link up status on the specified interface (index 1
   in IP instance list of interfaces). */
status = nx_ip_interface_status_check(&ip_0, 1, NX_IP_LINK_ENABLED,
                                      &actual_status, 10);

/* If status is NX_SUCCESS, the link for the specified interface is up. */
```

See Also

nx_ip_address_change_notify, nx_ip_ address_get, nx_ip_ address_set, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_info_get, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check

nx_ip_raw_packet_disable

Disable raw packet sending/receiving

Prototype

```
UINT nx_ip_raw_packet_disable(NX_IP *ip_ptr);
```

Description

This service disables transmission and reception of raw IP packets for this IP instance.

Parameters

ip_ptr	Pointer to previously created IP instance.

Return Values

NX_SUCCESS	(0x00)	Successful IP raw packet disable.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads, timers

Preemption Possible

No

Example

```
/* Disable raw packet sending/receiving for this IP instance.  */
status = nx_ip_raw_packet_disable(&ip_0);

/* If status is NX_SUCCESS, raw IP packet sending/receiving has been
   disabled for the previously created IP instance.  */
```

See Also

nx_ip_address_change_notify, nx_ip_address_get, nx_ip_address_set, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_info_get, nx_ip_raw_packet_enable, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check

nx_ip_raw_packet_enable

Enable raw packet sending/receiving

Prototype

```
UINT nx_ip_raw_packet_enable(NX_IP *ip_ptr);
```

Description

This service enables transmission and reception of raw IP packets for this IP instance. By enabling raw IP facilities, the protocol field in the IP header is not examined on reception. Instead, all incoming IP packets are placed on the raw IP receive queue.

Parameters

ip_ptr	Pointer to previously created IP instance.

Return Values

NX_SUCCESS	(0x00)	Successful IP raw packet enable.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads, timers

Preemption Possible

No

Example

```
/* Enable raw packet sending/receiving for this IP instance.  */
status = nx_ip_raw_packet_enable(&ip_0);

/* If status is NX_SUCCESS, raw IP packet sending/receiving has been
   enabled for the previously created IP instance.  */
```

See Also

nx_ip_address_change_notify, nx_ip_address_change_notify, nx_ip_address_get, nx_ip_address_set, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_info_get, nx_ip_raw_packet_disable, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check

nx_ip_raw_packet_interface_send

Send raw IP packet out specified network interface

Prototype

```
UINTnx_ip_raw_packet_interface_send(NX_IP*ip_ptr,NX_PACKET*packet_ptr,
                                    ULONG destination_ip,
                                    UINT interface_index,
                                    ULONG type_of_service);
```

Description

This function sends a raw IP packet through the specified network interface.

Note that raw IP processing must be enabled.

Parameters

ip_ptr	Pointer to previously created IP task.
packet_ptr	Pointer to packet to transmit.
destination_ip	IP address to send packet.
interface_index	Index of interface to send packet out on.
type_of_service	Type of service for packet.

Return Values

NX_SUCCESS	(0x00)	Packet successfully transmitted.
NX_IP_ADDRESS_ERROR	(0x21)	No suitable outgoing interface available.
NX_NOT_ENABLED	(0x14)	Raw IP packet processing not enabled.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_PTR_ERROR	(0x07)	Invalid pointer input.
NX_OPTION_ERROR	(0x0A)	Invalid type of service specified.
NX_OVERFLOW	(0x03)	Invalid packet prepend pointer.
NX_UNDERFLOW	(0x02)	Invalid packet prepend pointer.
NX_INVALID_INTERFACE	(0x4C)	Invalid interface index specified.

Allowed From

Threads

Preemption Possible

Yes

Example

```
/* Send packet out on interface 1 with normal type of service.  */
status = nx_ip_raw_packet_interface_send( ip_ptr, packet_ptr,
                                          destination_ip, 1, NX_IP_NORMAL);

/* If status is NX_SUCCESS the packet was successfully transmitted. */
```

See Also

nx_ip_address_change_notify, nx_ip_ address_get, nx_ip_ address_set, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_info_get, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check

nx_ip_raw_packet_receive

Receive raw IP packet

Prototype

```
UINT nx_ip_raw_packet_receive(NX_IP *ip_ptr,
                              NX_PACKET **packet_ptr, ULONG wait_option);
```

Description

This service receives a raw IP packet from the specified IP instance. If there are IP packets on the raw packet receive queue, the first (oldest) packet is returned to the caller. Otherwise, if no packets are available, the caller may suspend as specified by the wait option.

If NX_SUCCESS, is returned, the application is responsible for releasing the received packet when it is no longer needed.

Parameters

ip_ptr	Pointer to previously created IP instance.
packet_ptr	Pointer to pointer to place the received raw IP packet in.
wait_option	Defines how the service behaves if there are no raw IP packets available. The wait options are defined as follows: NX_NO_WAIT (0x00000000) NX_WAIT_FOREVER (0xFFFFFFFF) timeout value (0x00000001 through 0xFFFFFFFE)

Return Values

NX_SUCCESS	(0x00)	Successful IP raw packet receive.
NX_NO_PACKET	(0x01)	No packet was available.
NX_WAIT_ABORTED	(0x1A)	Requested suspension was aborted by a call to *tx_thread_wait_abort*.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

NX_PTR_ERROR	(0x07)	Invalid IP or return packet pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service

Allowed From

Threads

Preemption Possible

Yes

Example

```
/* Receive a raw IP packet for this IP instance, wait for a maximum
   of 4 timer ticks.  */
status = nx_ip_raw_packet_receive(&ip_0, &packet_ptr, 4);

/* If status is NX_SUCCESS, the raw IP packet pointer is in the
   variable packet_ptr.  */
```

See Also

nx_ip_address_change_notify, nx_ip_address_change_notify, nx_ip_address_get, nx_ip_address_set, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_enable, nx_ip_fragment_disable, nx_ip_gateway_address_set, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_info_get, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check

nx_ip_raw_packet_send

Send raw IP packet

Prototype

```
UINT nx_ip_raw_packet_send(NX_IP *ip_ptr,
                           NX_PACKET *packet_ptr, ULONG destination_ip,
                           ULONG type_of_service);
```

Description

This service sends a raw IP packet to the specified destination IP address. Note that this routine returns immediately, and it is therefore not known whether the IP packet has actually been sent. The network driver will be responsible for releasing the packet when the transmission is complete. This service differs from other services in that there is no way of knowing if the packet was actually sent. It could get lost on the Internet.

Unless an error is returned, the application should not release the packet after this call. Doing so will cause unpredictable results because the network driver will release the packet after transmission.

Parameters

ip_ptr	Pointer to previously created IP instance.
packet_ptr	Pointer to the raw IP packet to send.
destination_ip	Destination IP address, which can be a specific host IP address, a network broadcast, an internal loop-back, or a multicast address.
type_of_service	Defines the type of service for the transmission, legal values are as follows:

NX_IP_NORMAL	(0x00000000)
NX_IP_MIN_DELAY	(0x00100000)
NX_IP_MAX_DATA	(0x00080000)
NX_IP_MAX_RELIABLE	(0x00040000)
NX_IP_MIN_COST	(0x00020000)

Return Values

NX_SUCCESS	(0x00)	Successful IP raw packet send initiated.
NX_NOT_ENABLED	(0x14)	Raw IP feature is not enabled.
NX_IP_ADDRESS_ERROR	(0x21)	Invalid IP address.
NX_OPTION_ERROR	(0x0a)	Invalid type of service.
NX_UNDERFLOW	(0x02)	Not enough room to prepend an IP header on the packet.
NX_OVERFLOW	(0x03)	Packet append pointer is invalid.
NX_PTR_ERROR	(0x07)	Invalid IP or packet pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Threads

Preemption Possible

Yes

Example

```
/* Send a raw IP packet to IP address 1.2.3.5.  */
status = nx_ip_raw_packet_send(&ip_0, packet_ptr,
                        IP_ADDRESS(1,2,3,5),
                        NX_IP_NORMAL);

/* If status is NX_SUCCESS, the raw IP packet pointed to by
   packet_ptr has been sent.  */
```

See Also

nx_ip_address_change_notify, nx_ip_address_get, nx_ip_address_set, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_info_get, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_receive, nx_ip_static_route_add, nx_ip_static_route_delete, nx_ip_status_check

nx_ip_static_route_add

Add static route

Prototype

```
UINT nx_ip_static_route_add(NX_IP *ip_ptr, ULONG network_address,
                            ULONG net_mask, ULONG next_hop)
```

Description

This function adds an entry to the static routing table. Note that the *next_hop* address must be directly accessible from the local interface.

Note that ip_ptr must point to a valid NetX IP structure and static routing must be enabled via NX_ENABLE_IP_STATIC_ROUTING to use this service.

Parameters

ip_ptr	Pointer to previously created IP instance.
network_address	Target network address, in host byte order
net_mask	Target network mask, in host byte order
next_hop	Next hop address for the target network, in host byte order

Return Values

NX_SUCCESS	(0x00)	Entry is added to the static routing table.
NX_OVERFLOW	(0x03)	Static routing table is full.
NX_NOT_IMPLEMENTED	(0x4A)	This feature is not compiled in.
NX_IP_ADDRESS_ERROR	(0x21)	Next hop is not directly accessible via local interfaces.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_PTR_ERROR	(0x07)	Invalid *ip_ptr* pointer.

Allowed From

Initialization, threads

Preemption Possible

No

Example

```
/* Specify the next hop for 192.168.1.68 through the gateway
   192.168.1.1.  */
status =  nx_ip_static_route_add(ip_ptr, IP_ADDRESS(192,168,1,68),
                                 0xFFFFFF00UL, IP_ADDRESS(192,168,1,1));

/* If status is NX_SUCCESS the route was successfully added to the static
   routing table.  */
```

See Also

nx_ip_address_change_notify, nx_ip_ address_get, nx_ip_ address_set, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_info_get, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_delete, nx_ip_status_check

nx_ip_static_route_delete

Delete static route

Prototype

```
UINT nx_ip_static_route_delete(NX_IP *ip_ptr, ULONG network_address,
                               ULONG net_mask);
```

Description

This function deletes an entry from the static routing table.

Note that ip_ptr must point to a valid NetX IP structure and static routing must be enabled via NX_ENABLE_IP_STATIC_ROUTING to use this service.

Parameters

ip_ptr	Pointer to previously created IP instance.
network_address	Target network address, in host byte order.
net_mask	Target network mask, in host byte order.

Return Values

NX_SUCCESS	(0x00)	Successful deletion from the static routing table.
NX_NOT_SUCCESSFUL	(0x43)	Entry cannot be found in the routing table.
NX_NOT_IMPLEMENTED	(0x4A)	This feature is not compiled in.
NX_PTR_ERROR	(0x07)	Invalid *ip_ptr* pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization,threads

Preemption Possible

No

Example

```
/* Remove the static route for 192.168.1.68 from the routing table.  */
status =  nx_ip_static_route_delete(ip_ptr, IP_ADDRESS(192,168,1,68),
                                                0xFFFFFF00UL,);

/* If status is NX_SUCCESS the route was successfully removed from the
   static routing table.  */
```

See Also

nx_ip_address_change_notify, nx_ip_ address_get, nx_ip_ address_set, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_info_get, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_status_check

nx_ip_status_check

Check status of an IP instance

Prototype

```
UINT nx_ip_status_check(NX_IP *ip_ptr, ULONG needed_status,
                        ULONG *actual_status, ULONG wait_option);
```

Description

This service checks and optionally waits for the specified status of a previously created IP instance.

Parameters

ip_ptr	Pointer to previously created IP instance.
needed_status	IP status requested, defined in bit-map form as follows: NX_IP_INITIALIZE_DONE (0x0001) NX_IP_ADDRESS_RESOLVED (0x0002) NX_IP_LINK_ENABLED (0x0004) NX_IP_ARP_ENABLED (0x0008) NX_IP_UDP_ENABLED (0x0010) NX_IP_TCP_ENABLED (0x0020) NX_IP_IGMP_ENABLED (0x0040) NX_IP_RARP_COMPLETE (0x0080)
actual_status	Pointer to destination of actual bits set.
wait_option	Defines how the service behaves if the requested status bits are not available. The wait options are defined as follows: NX_NO_WAIT 0x00000000) timeout value (0x00000001 through 0xFFFFFFFE)

Return Values

NX_SUCCESS	(0x00)	Successful IP status check.
NX_PTR_ERROR	(0x07)	IP pointer is or has become invalid, or actual status pointer is invalid.
NX_NOT_SUCCESSFUL	(0x43)	Status request was not satisfied within the timeout specified.
NX_OPTION_ERROR	(0x0a)	Invalid needed status option.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Threads

Preemption Possible

Yes

Example

```
/* Wait 10 ticks for the link up status on the previously created IP
   instance.   */
status = nx_ip_status_check(&ip_0, NX_IP_LINK_ENABLED,
                            &actual_status, 10);

/* If status is NX_SUCCESS, the link for the specified IP instance
   is up.  */
```

See Also

nx_ip_address_change_notify, nx_ip_address_get, nx_ip_address_set, nx_ip_create, nx_ip_delete, nx_ip_driver_direct_command, nx_ip_forwarding_disable, nx_ip_forwarding_enable, nx_ip_fragment_disable, nx_ip_fragment_enable, nx_ip_gateway_address_set, nx_ip_info_get, nx_ip_interface_address_get, nx_ip_interface_address_set, nx_ip_interface_attach, nx_ip_interface_info_get, nx_ip_interface_status_check, nx_ip_raw_packet_disable, nx_ip_raw_packet_enable, nx_ip_raw_packet_interface_send, nx_ip_raw_packet_receive, nx_ip_raw_packet_send, nx_ip_static_route_add, nx_ip_static_route_delete

nx_packet_allocate

Allocate packet from specified pool

Prototype

```
UINT nx_packet_allocate(NX_PACKET_POOL *pool_ptr,
                        NX_PACKET **packet_ptr, ULONG packet_type,
                        ULONG wait_option);
```

Description

This service allocates a packet from the specified pool and adjusts the prepend pointer in the packet according to the type of packet specified. If no packet is available, the service suspends according to the supplied wait option.

Parameters

pool_ptr	Pointer to previously created packet pool.
packet_ptr	Pointer to the pointer of the allocated packet pointer.
packet_type	Defines the type of packet requested, legal values are as follows: NX_IP_PACKET (0x24) NX_UDP_PACKET (0x2C) NX_TCP_PACKET (0x38) NX_RECEIVE_PACKET (0x00)
wait_option	Defines how the service behaves if there are no packets available. The wait options are defined as follows: NX_NO_WAIT (0x00000000) NX_WAIT_FOREVER (0xFFFFFFFF) timeout value (0x00000001 through 0xFFFFFFFE)

Return Values

NX_SUCCESS	(0x00)	Successful packet allocate.
NX_NO_PACKET	(0x01)	No packet available.
NX_WAIT_ABORTED	(0x1A)	Requested suspension was aborted by a call to *tx_thread_wait_abort*.
NX_OPTION_ERROR	(0x0A)	Invalid packet type.
NX_PTR_ERROR	(0x07)	Invalid pool or packet return pointer.
NX_INVALID_PARAMETERS	(0x4D)	Packet size cannot support protocol.
NX_CALLER_ERROR	(0x11)	Invalid wait option from non-thread.

Allowed From

Initialization, threads, timers, and ISRs (application network drivers)

Preemption Possible

Yes

Example

```
/* Allocate a new UDP packet from the previously created packet pool and
   suspend for a maximum of 5 timer ticks if the pool is empty.   */
status = nx_packet_allocate(&pool_0, &packet_ptr,
                            NX_UDP_PACKET, 5);

/* If status is NX_SUCCESS, the newly allocated packet pointer is found
   in the variable packet_ptr.  */
```

See Also

nx_packet_copy, nx_packet_data_append, nx_packet_data_extract_offset, nx_packet_data_retrieve, nx_packet_length_get, nx_packet_pool_create, nx_packet_pool_delete, nx_packet_pool_info_get, nx_packet_release, nx_packet_transmit_release

nx_packet_copy

Copy packet

Prototype

```
UINT  nx_packet_copy(NX_PACKET *packet_ptr,
                     NX_PACKET **new_packet_ptr,
                     NX_PACKET_POOL *pool_ptr, ULONG wait_option);
```

Description

This service copies the information in the supplied packet to one or more new packets that are allocated from the supplied packet pool. If successful, the pointer to the new packet is returned in destination pointed to by **new_packet_ptr**.

Parameters

packet_ptr	Pointer to the source packet.
new_packet_ptr	Pointer to the destination of where to return the pointer to the new copy of the packet.
pool_ptr	Pointer to the previously created packet pool that is used to allocate one or more packets for the copy.
wait_option	Defines how the service waits if there are no packets available. The wait options are defined as follows:

NX_NO_WAIT	(0x00000000)
NX_WAIT_FOREVER	(0xFFFFFFFF)
timeout value	(0x00000001 through 0xFFFFFFFE)

Return Values

NX_SUCCESS	(0x00)	Successful packet copy.
NX_NO_PACKET	(0x01)	Packet not available for copy.
NX_INVALID_PACKET	(0x12)	Empty source packet or copy failed.
NX_WAIT_ABORTED	(0x1A)	Requested suspension was aborted by a call to tx_thread_wait_abort.

NX_PTR_ERROR	(0x07)	Invalid pool, packet, or destination pointer.
NX_UNDERFLOW	(0x02)	Invalid packet prepend pointer.
NX_OVERFLOW	(0x03)	Invalid packet append pointer.
NX_CALLER_ERROR	(0x11)	A wait option was specified in initialization or in an ISR.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

Yes

Example

```
NX_PACKET               *new_copy_ptr;

/* Copy packet pointed to by "old_packet_ptr" using packets from
   previously created packet pool_0. */
status = nx_packet_copy(old_packet, &new_copy_ptr, &pool_0, 20);

/* If status is NX_SUCCESS, new_copy_ptr points to the packet copy. */
```

See Also

nx_packet_allocate, nx_packet_data_append, nx_packet_data_extract_offset, nx_packet_data_retrieve, nx_packet_length_get, nx_packet_pool_create, nx_packet_pool_delete, nx_packet_pool_info_get, nx_packet_release, nx_packet_transmit_release

nx_packet_data_append

Append data to end of packet

Prototype

```
UINT nx_packet_data_append(NX_PACKET *packet_ptr,
                           VOID *data_start, ULONG data_size,
                           NX_PACKET_POOL *pool_ptr,
                           ULONG wait_option);
```

Description

This service appends data to the end of the specified packet. The supplied data area is copied into the packet. If there is not enough memory available, one or more packets will be allocated to satisfy the request.

Parameters

packet_ptr	Packet pointer.
data_start	Pointer to the start of the user's data area to append to the packet.
data_size	Size of user's data area.
pool_ptr	Pointer to packet pool from which to allocate another packet if there is not enough room in the current packet.
wait_option	Defines how the service behaves if there are no packets available. The wait options are defined as follows:

NX_NO_WAIT	(0x00000000)
NX_WAIT_FOREVER	(0xFFFFFFFF)
timeout value	(0x00000001 through 0xFFFFFFFE)

Return Values

NX_SUCCESS	(0x00)	Successful packet append.
NX_NO_PACKET	(0x01)	No packet available.
NX_WAIT_ABORTED	(0x1A)	Requested suspension was aborted by a call to *tx_thread_wait_abort*.
NX_UNDERFLOW	(0x02)	Prepend pointer is less than payload start.
NX_OVERFLOW	(0x03)	Append pointer is greater than payload end.
NX_PTR_ERROR	(0x07)	Invalid pool, packet, or data Pointer.
NX_SIZE_ERROR	(0x09)	Invalid data size.
NX_CALLER_ERROR	(0x11)	Invalid wait option from non-thread.

Allowed From

Initialization, threads, timers, and ISRs (application network drivers)

Preemption Possible

Yes

Example

```
/* Append "abcd" to the specified packet.  */
status = nx_packet_data_append(packet_ptr, "abcd", 4, &pool_0, 5);

/* If status is NX_SUCCESS, the additional four bytes "abcd" have
   been appended to the packet.  */
```

See Also

nx_packet_allocate, nx_packet_copy, nx_packet_data_extract_offset, nx_packet_data_retrieve, nx_packet_length_get, nx_packet_pool_create, nx_packet_pool_delete, nx_packet_pool_info_get, nx_packet_release, nx_packet_transmit_release

nx_packet_data_extract_offset

Extract data from packet via an offset

Prototype

```
UINT nx_packet_data_extract_offset( NX_PACKET *packet_ptr, ULONG offset,
                                    VOID *buffer_start,
                                    ULONG buffer_length,
                                    ULONG *bytes_copied);
```

Description

This service copies data from a NetX packet (or packet chain) starting at the specified offset from the packet prepend pointer of the specified size in bytes into the specified buffer. The number of bytes actually copied is returned in *bytes_copied.* This service does not remove data from the packet, nor does it adjust the prepend pointer.

Parameters

packet_ptr	Pointer to packet to extract
offset	Offset from the current prepend pointer.
buffer_start	Pointer to start of save buffer
buffer_length	Number of bytes to copy
bytes_copied	Number of bytes actually copied

Return Values

NX_SUCCESS	(0x00)	Successful packet copy
NX_PTR_ERROR	(0x07)	Invalid packet pointer or buffer pointer
NX_PACKET_OFFSET_ERROR	(0x43)	Invalid offset value was supplied

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
/* Extract 10 bytes from the start of the received packet buffer into the
   specified memory area. */
status = nx_packet_data_extract_offset(my_packet, 0, &data[0], 10,
                                                     &bytes_copied) ;

/* If status is NX_SUCCESS, 10 bytes were successfully copied into the data
   buffer. */
```

See Also

nx_packet_allocate, nx_packet_copy, nx_packet_data_append, nx_packet_data_extract_offset, nx_packet_data_retrieve, nx_packet_length_get, nx_packet_pool_create, nx_packet_pool_delete, nx_packet_pool_info_get, nx_packet_release, nx_packet_transmit_release

nx_packet_data_retrieve

Retrieve data from packet

Prototype

```
UINT  nx_packet_data_retrieve(NX_PACKET *packet_ptr, VOID *buffer_start,
                              ULONG *bytes_copied);
```

Description

This service copies data from the supplied packet into the supplied buffer. The actual number of bytes copied is returned in the destination pointed to by **bytes_copied**.

The destination buffer must be large enough to hold the packet's contents. If not, memory will be corrupted causing unpredictable results.

Parameters

packet_ptr	Pointer to the source packet.
buffer_start	Pointer to the start of the buffer area.
bytes_copied	Pointer to the destination for the number of bytes copied.

Return Values

NX_SUCCESS	(0x00)	Successful packet data retrieve.
NX_INVALID_PACKET	(0x12)	Invalid packet.
NX_PTR_ERROR	(0x07)	Invalid packet, buffer start, or bytes copied pointer.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
UCHAR                 buffer[512];
ULONG                 bytes_copied;

/* Retrieve data from packet pointed to by "packet_ptr". */
status = nx_packet_data_retrieve(packet_ptr, buffer, &bytes_copied);

/* If status is NX_SUCCESS, buffer contains the contents of the
   packet, the size of which is contained in "bytes_copied." */
```

See Also

nx_packet_allocate, nx_packet_copy, nx_packet_data_append, nx_packet_data_extract_offset, nx_packet_length_get, nx_packet_pool_create, nx_packet_pool_delete, nx_packet_pool_info_get, nx_packet_release, nx_packet_transmit_release

nx_packet_length_get

Get length of packet data

Prototype

```
UINT nx_packet_length_get (NX_PACKET *packet_ptr, ULONG *length);
```

Description

This service gets the length of the data in the specified packet.

Parameters

packet_ptr	Pointer to the packet.
length	Destination for the packet length.

Return Values

NX_SUCCESS	(0x00)	Successful packet length get.
NX_PTR_ERROR	(0x07)	Invalid packet pointer.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
/* Get the length of the data in "my_packet." */
status = nx_packet_length_get(my_packet, &my_length);

/* If status is NX_SUCCESS, data length is in "my_length". */
```

See Also

nx_packet_allocate, nx_packet_copy, nx_packet_data_append, nx_packet_data_extract_offset, nx_packet_data_retrieve, nx_packet_pool_create, nx_packet_pool_delete, nx_packet_pool_info_get, nx_packet_release, nx_packet_transmit_release

nx_packet_pool_create

Create packet pool in specified memory area

Prototype

```
UINT nx_packet_pool_create(NX_PACKET_POOL *pool_ptr,
                           CHAR *name, ULONG payload_size,
                           VOID *memory_ptr, ULONG memory_size);
```

Description

This service creates a packet pool of the specified packet size in the memory area supplied by the user.

Parameters

pool_ptr	Pointer to packet pool control block.
name	Pointer to application's name for the packet pool.
payload_size	Number of bytes in each packet in the pool. This value must be at least 40 bytes and must also be evenly divisible by 4.
memory_ptr	Pointer to the memory area to place the packet pool in. The pointer should be aligned on an ULONG boundary.
memory_size	Size of the pool memory area.

Return Values

NX_SUCCESS	(0x00)	Successful packet pool create.
NX_PTR_ERROR	(0x07)	Invalid pool or memory pointer.
NX_SIZE_ERROR	(0x09)	Invalid block or memory size.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads

Preemption Possible

No

Example

```
/* Create a packet pool of 32000 bytes starting at physical
   address 0x10000000.  */
status = nx_packet_pool_create(&pool_0, "Default Pool", 128,
                        (void *) 0x10000000, 32000);

/* If status is NX_SUCCESS, the packet pool has been successfully
   created.  */
```

See Also

nx_packet_allocate, nx_packet_copy, nx_packet_data_append, nx_packet_data_extract_offset, nx_packet_data_retrieve, nx_packet_length_get, nx_packet_pool_delete, nx_packet_pool_info_get, nx_packet_release, nx_packet_transmit_release

nx_packet_pool_delete

Delete previously created packet pool

Prototype

```
UINT nx_packet_pool_delete(NX_PACKET_POOL *pool_ptr);
```

Description

This service deletes a previously create packet pool.

Parameters

pool_ptr	Packet pool control block pointer.

Return Values

NX_SUCCESS	(0x00)	Successful packet pool delete.
NX_PTR_ERROR	(0x07)	Invalid pool pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Threads

Preemption Possible

No

Example

```
/* Delete a previously created packet pool.  */
status = nx_packet_pool_delete(&pool_0);

/* If status is NX_SUCCESS, the packet pool has been successfully
   deleted.  */
```

See Also

nx_packet_allocate, nx_packet_copy, nx_packet_data_append, nx_packet_data_extract_offset, nx_packet_data_retrieve, nx_packet_length_get, nx_packet_pool_create, nx_packet_pool_info_get, nx_packet_release, nx_packet_transmit_release

nx_packet_pool_info_get

Retrieve information about a packet pool

Prototype

```
UINT nx_packet_pool_info_get(NX_PACKET_POOL *pool_ptr,
                             ULONG *total_packets,
                             ULONG *free_packets,
                             ULONG *empty_pool_requests,
                             ULONG *empty_pool_suspensions,
                             ULONG *invalid_packet_releases);
```

Description

This service retrieves information about the specified packet pool.

If a destination pointer is NX_NULL, that particular information is not returned to the caller.

Parameters

pool_ptr	Pointer to previously created packet pool.
total_packets	Pointer to destination for the total number of packets in the pool.
free_packets	Pointer to destination for the total number of currently free packets.
empty_pool_requests	Pointer to destination of the total number of allocation requests when the pool was empty.
empty_pool_suspensions	Pointer to destination of the total number of empty pool suspensions.
invalid_packet_releases	Pointer to destination of the total number of invalid packet releases.

Return Values

NX_SUCCESS	(0x00)	Successful packet pool information retrieval.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads, and timers

Preemption Possible

No

Example

```
/* Retrieve packet pool information.  */
status =  nx_packet_pool_info_get(&pool_0,
                        &total_packets,
                        &free_packets,
                        &empty_pool_requests,
                        &empty_pool_suspensions,
                        &invalid_packet_releases);

/* If status is NX_SUCCESS, packet pool information was retrieved. */
```

See Also

nx_packet_allocate, nx_packet_copy, nx_packet_data_append, nx_packet_data_extract_offset, nx_packet_data_retrieve, nx_packet_length_get, nx_packet_pool_create, nx_packet_pool_delete, nx_packet_release, nx_packet_transmit_release

nx_packet_release

Release previously allocated packet

Prototype

```
UINT nx_packet_release(NX_PACKET *packet_ptr);
```

Description

This service releases a packet, including any additional packets linked to the specified packet. If another thread is waiting for packets, it is given the packet and resumed.

The application must prevent releasing a packet more than once, because doing so will cause unpredictable results.

Parameters

packet_ptr	Packet pointer.

Return Values

NX_SUCCESS	(0x00)	Successful packet release.
NX_PTR_ERROR	(0x07)	Invalid packet pointer.

Allowed From

Initialization, threads, timers, and ISRs (application network drivers)

Preemption Possible

Yes

Example

```
/* Release a previously allocated packet.  */
status = nx_packet_release(packet_ptr);

/* If status is NX_SUCCESS, the packet has been returned to the
   packet pool it was allocated from.  */
```

See Also

nx_packet_allocate, nx_packet_copy, nx_packet_data_append, nx_packet_data_extract_offset, nx_packet_data_retrieve, nx_packet_length_get, nx_packet_pool_create, nx_packet_pool_delete, nx_packet_pool_info_get, nx_packet_transmit_release

nx_packet_transmit_release

Release a transmitted packet

Prototype

```
UINT nx_packet_transmit_release(NX_PACKET *packet_ptr);
```

Description

This service releases a transmitted packet, including any additional packets linked to the specified packet. If another thread is waiting for packets, it is given the packet and resumed. This routine is typically called from the application's network driver.

The network driver should remove the physical media header and adjust the length of the packet before calling this service.

Parameters

packet_ptr	Packet pointer.

Return Values

NX_SUCCESS	(0x00)	Successful transmit packet release.
NX_PTR_ERROR	(0x07)	Invalid packet pointer.

Allowed From

Application network drivers (including ISRs)

Preemption Possible

Yes

Example

```
/* Release a previously allocated packet that was just transmitted
   from the application network driver.  */
status = nx_packet_transmit_release(packet_ptr);

/* If status is NX_SUCCESS, the transmitted packet has been returned
   to the packet pool it was allocated from.  */
```

See Also

nx_packet_allocate, nx_packet_copy, nx_packet_data_append, nx_packet_data_extract_offset, nx_packet_data_retrieve, nx_packet_length_get, nx_packet_pool_create, nx_packet_pool_delete, nx_packet_pool_info_get, nx_packet_release

nx_rarp_disable

Disable Reverse Address Resolution Protocol (RARP)

Prototype

```
UINT nx_rarp_disable(NX_IP *ip_ptr);
```

Description

This service disables the RARP component of NetX for the specific IP instance.

Parameters

ip_ptr	Pointer to previously created IP instance.

Return Values

NX_SUCCESS	(0x00)	Successful RARP disable.
NX_NOT_ENABLED	(0x14)	RARP was not enabled.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads, timers

Preemption Possible

No

Example

```
/* Disable RARP on the previously created IP instance.  */
status = nx_rarp_disable(&ip_0);

/* If status is NX_SUCCESS, RARP is disabled.  */
```

See Also

nx_rarp_enable, nx_rarp_info_get

nx_rarp_enable

Enable Reverse Address Resolution Protocol (RARP)

Prototype

```
UINT nx_rarp_enable(NX_IP *ip_ptr);
```

Description

This service enables the RARP component of NetX for the specific IP instance. Note that the IP instance must be created with an IP address of zero in order to use RARP. A non-zero IP address implies that it is valid.

Parameters

ip_ptr	Pointer to previously created IP instance.

Return Values

NX_SUCCESS	(0x00)	Successful RARP enable.
NX_IP_ADDRESS_ERROR	(0x21)	IP address is already valid.
NX_ALREADY_ENABLED	(0x15)	RARP was already enabled.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads, timers

Preemption Possible

No

Example

```
/* Enable RARP on the previously created IP instance.  */
status = nx_rarp_enable(&ip_0);

/* If status is NX_SUCCESS, RARP is enabled and is attempting to
   resolve this IP instance's address by querying the network.  */
```

See Also

nx_rarp_disable, nx_rarp_info_get

nx_rarp_info_get

Retrieve information about RARP activities

Prototype

```
UINT nx_rarp_info_get(NX_IP *ip_ptr,
                      ULONG *rarp_requests_sent,
                      ULONG *rarp_responses_received,
                      ULONG *rarp_invalid_messages);
```

Description

This service retrieves information about RARP activities for the specified IP instance.

If a destination pointer is NX_NULL, that particular information is not returned to the caller.

Parameters

ip_ptr	Pointer to previously created IP instance.
rarp_requests_sent	Pointer to destination for the total number of RARP requests sent.
rarp_responses_received	Pointer to destination for the total number of RARP responses received.
rarp_invalid_messages	Pointer to destination of the total number of invalid messages.

Return Values

NX_SUCCESS	(0x00)	Successful RARP information retrieval.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads, and timers

Preemption Possible

No

Example

```
/* Retrieve RARP information from previously created IP Instance 0. */
status =  nx_rarp_info_get(&ip_0,
                           &rarp_requests_sent,
                           &rarp_responses_received,
                           &rarp_invalid_messages);

/* If status is NX_SUCCESS, RARP information was retrieved.  */
```

See Also

nx_rarp_disable, nx_rarp_enable

nx_system_initialize

Initialize NetX System

Prototype

```
VOID nx_system_initialize(VOID);
```

Description

This service initializes the basic NetX system resources in preparation for use. It should be called by the application during initialization and before any other NetX call are made.

Parameters

None

Return Values

None

Allowed From

Initialization, threads

Preemption Possible

No

Example

```
/* Initialize NetX for operation.  */
nx_system_initialize();

/* At this point, NetX is ready for IP creation and all subsequent
   network operations.  */
```

See Also

None

nx_tcp_client_socket_bind

Bind client TCP socket to TCP port

Prototype

```
UINT nx_tcp_client_socket_bind(NX_TCP_SOCKET *socket_ptr,
                               UINT port, ULONG wait_option);
```

Description

This service binds the previously created TCP client socket to the specified TCP port. Valid TCP sockets range from 0 through 0xFFFF.

Parameters

socket_ptr	Pointer to previously created TCP socket instance.
port	Number of port to bind (1 through 0xFFFF). If port number is NX_ANY_PORT (0x0000), the IP instance will search for the next free port and use that for the binding.
wait_option	Defines how the service behaves if the port is already bound to another socket. The wait options are defined as follows: NX_NO_WAIT (0x00000000) NX_WAIT_FOREVER (0xFFFFFFFF) timeout value (0x00000001 through 0xFFFFFFFE)

Return Values

NX_SUCCESS	(0x00)	Successful socket bind.
NX_ALREADY_BOUND	(0x22)	This socket is already bound to another TCP port.
NX_PORT_UNAVAILABLE	(0x23)	Port is already bound to a different socket.
NX_NO_FREE_PORTS	(0x45)	No free port.
NX_WAIT_ABORTED	(0x1A)	Requested suspension was aborted by a call to *tx_thread_wait_abort*.
NX_INVALID_PORT	(0x46)	Invalid port.

NX_PTR_ERROR	(0x07)	Invalid socket pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

Yes

Example

```
/* Bind a previously created client socket to port 12 and wait for 7
   timer ticks for the bind to complete.  */
status = nx_tcp_client_socket_bind(&client_socket, 12, 7);

/* If status is NX_SUCCESS, the previously created client_socket is
   bound to port 12 on the associated IP instance.  */
```

See Also

nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_client_socket_connect

Connect client TCP socket

Prototype

```
UINT nx_tcp_client_socket_connect(NX_TCP_SOCKET *socket_ptr,
                          UINT server_ip, UINT server_port,
                          ULONG wait_option)
```

Description

This service connects the previously created TCP client socket to the specified server's port. Valid TCP server ports range from 0 through 0xFFFF.

Parameters

socket_ptr	Pointer to previously created TCP socket instance.
server_ip	Server's IP address.
server_port	Server port number to connect to (1 through 0xFFFF).
wait_option	Defines how the service behaves while the connection is being established. The wait options are defined as follows: NX_NO_WAIT (0x00000000) NX_WAIT_FOREVER (0xFFFFFFFF) timeout value (0x00000001 through 0xFFFFFFFE)

Return Values

NX_SUCCESS	(0x00)	Successful socket connect.
NX_NOT_BOUND	(0x24)	Socket is not bound.
NX_NOT_CLOSED	(0x35)	Socket is not in a closed state.
NX_IN_PROGRESS	(0x37)	No wait was specified, the connection attempt is in progress.
NX_WAIT_ABORTED	(0x1A)	Requested suspension was aborted by a call to *tx_thread_wait_abort*.

NX_IP_ADDRESS_ERROR	(0x21)	Invalid server IP address.
NX_INVALID_PORT	(0x46)	Invalid port.
NX_PTR_ERROR	(0x07)	Invalid socket pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

Yes

Example

```
/* Initiate a TCP connection from a previously created and bound
   client socket. The connection requested in this example is to
   port 12 on the server with the IP address of 1.2.3.5. This
   service will wait 300 timer ticks for the connection to take
   place before giving up.  */
status = nx_tcp_client_socket_connect(&client_socket,
                      IP_ADDRESS(1,2,3,5), 12, 300);

/* If status is NX_SUCCESS, the previously created and bound
   client_socket is connected to port 12 on IP 1.2.3.5.  */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_client_socket_port_get

Get port number bound to client TCP socket

Prototype

```
UINT nx_tcp_client_socket_port_get(NX_TCP_SOCKET *socket_ptr,
                                   UINT *port_ptr);
```

Description

This service retrieves the port number associated with the socket, which is useful to find the port allocated by NetX in situations where the NX_ANY_PORT was specified at the time the socket was bound.

Parameters

socket_ptr	Pointer to previously created TCP socket instance.
port_ptr	Pointer to destination for the return port number. Valid port numbers are (1 through 0xFFFF).

Return Values

NX_SUCCESS	(0x00)	Successful socket bind.
NX_NOT_BOUND	(0x24)	This socket is not bound to a port.
NX_PTR_ERROR	(0x07)	Invalid socket pointer or port return pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

No

Example

```
/* Get the port number of previously created and bound client
   socket.  */
status = nx_tcp_client_socket_port_get(&client_socket, &port);

/* If status is NX_SUCCESS, the port variable contains the port this
   socket is bound to.  */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_unbind, nx_tcp_enable,nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, , nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_client_socket_unbind

Unbind TCP client socket from TCP port

Prototype

```
UINT nx_tcp_client_socket_unbind(NX_TCP_SOCKET *socket_ptr);
```

Description

This service releases the binding between the TCP client socket and a TCP port. If there are other threads waiting to bind another socket to the unbound port, the first suspended thread is then bound to the newly unbound port.

Parameters

socket_ptr	Pointer to previously created TCP socket instance.

Return Values

NX_SUCCESS	(0x00)	Successful socket unbind.
NX_NOT_BOUND	(0x24)	Socket was not bound to any port.
NX_NOT_CLOSED	(0x35)	Socket has not been disconnected.
NX_PTR_ERROR	(0x07)	Invalid socket pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

Yes

Example

```
/* Unbind a previously created and bound client TCP socket.
status = nx_tcp_client_socket_unbind(&client_socket);

/* If status is NX_SUCCESS, the client socket is no longer bound.  */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_enable

Enable TCP component of NetX

Prototype

```
UINT nx_tcp_enable(NX_IP *ip_ptr);
```

Description

This service enables the Transmission Control Protocol (TCP) component of NetX. After enabled, TCP data may be sent and received by the application.

Parameters

ip_ptr	Pointer to previously created IP instance instance.

Return Values

NX_SUCCESS	(0x00)	Successful TCP enable.
NX_ALREADY_ENABLED	(0x15)	TCP is already enabled.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Initialization, threads, timers

Preemption Possible

No

Example

```
/* Enable TCP on a previously created IP instance. /*
status = nx_tcp_enable(&ip_0);

/* If status is NX_SUCCESS, TCP is enabled on the IP instance.  */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_free_port_find

Find next available TCP port

Prototype

```
UINT nx_tcp_free_port_find(NX_IP *ip_ptr, UINT port,
                           UINT *free_port_ptr);
```

Description

This service attempts to locate a free TCP port (unbound) starting from the application supplied port. The search logic will wrap around if the search happens to reach the maximum port value of 0xFFFF. If the search is successful, the free port is returned in the variable pointed to by ***free_port_ptr***.

This service can be called from another thread and have the same port returned. To prevent this race condition, the application may wish to place this service and the actual client socket bind under the protection of a mutex.

Parameters

ip_ptr	Pointer to previously created IP instance.
port	Port number to start search at (1 through 0xFFFF).
free_port_ptr	Pointer to the destination free port return value.

Return Values

NX_SUCCESS	(0x00)	Successful free port find.
NX_NO_FREE_PORTS	(0x45)	No free ports found.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.
NX_INVALID_PORT	(0x46)	The specified port number is invalid.

Allowed From

Threads

Preemption Possible

No

Example

```
/* Locate a free TCP port, starting at port 12, on a previously
   created IP instance.  */
status = nx_tcp_free_port_find(&ip_0, 12, &free_port);

/* If status is NX_SUCCESS, "free_port" contains the next free port
   on the IP instance.  */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_info_get

Retrieve information about TCP activities

Prototype

```
UINT nx_tcp_info_get(NX_IP *ip_ptr,
                     ULONG *tcp_packets_sent,
                     ULONG *tcp_bytes_sent,
                     ULONG *tcp_packets_received,
                     ULONG *tcp_bytes_received,
                     ULONG *tcp_invalid_packets,
                     ULONG *tcp_receive_packets_dropped,
                     ULONG *tcp_checksum_errors,
                     ULONG *tcp_connections,
                     ULONG *tcp_disconnections,
                     ULONG *tcp_connections_dropped,
                     ULONG *tcp_retransmit_packets);
```

Description

This service retrieves information about TCP activities for the specified IP instance.

If a destination pointer is NX_NULL, that particular information is not returned to the caller.

Parameters

ip_ptr	Pointer to previously created IP instance.
tcp_packets_sent	Pointer to destination for the total number of TCP packets sent.
tcp_bytes_sent	Pointer to destination for the total number of TCP bytes sent.
tcp_packets_received	Pointer to destination of the total number of TCP packets received.
tcp_bytes_received	Pointer to destination of the total number of TCP bytes received.
tcp_invalid_packets	Pointer to destination of the total number of invalid TCP packets.
tcp_receive_packets_dropped	Pointer to destination of the total number of TCP receive packets dropped.

tcp_checksum_errors	Pointer to destination of the total number of TCP packets with checksum errors.
tcp_connections	Pointer to destination of the total number of TCP connections.
tcp_disconnections	Pointer to destination of the total number of TCP disconnections.
tcp_connections_dropped	Pointer to destination of the total number of TCP connections dropped.
tcp_retransmit_packets	Pointer to destination of the total number of TCP packets retransmitted.

Return Values

NX_SUCCESS	(0x00)	Successful TCP information retrieval.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Initialization, threads, and timers

Preemption Possible

No

Example

```
/* Retrieve TCP information from previously created IP Instance 0.
   */
status =  nx_tcp_info_get(&ip_0,
                          &tcp_packets_sent,
                          &tcp_bytes_sent,
                          &tcp_packets_received,
                          &tcp_bytes_received,
                          &tcp_invalid_packets,
                          &tcp_receive_packets_dropped,
                          &tcp_checksum_errors,
                          &tcp_connections,
                          &tcp_disconnections
                          &tcp_connections_dropped,
                          &tcp_retransmit_packets);

/* If status is NX_SUCCESS, TCP information was retrieved.  */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

N E T X

nx_tcp_server_socket_accept

Accept TCP server connection

Prototype

```
UINT nx_tcp_server_socket_accept(NX_TCP_SOCKET *socket_ptr,
                                 ULONG wait_option);
```

Description

This service accepts (or prepares to accept) a TCP client socket connection request for a port that was previously set up for listening. This service may be called immediately after the application calls the listen or re-listen service or after the listen callback routine is called when the client connection is actually present.

*The application must call **nx_tcp_server_socket_unaccept** after the connection is no longer needed to remove the server socket's binding to the server port.*

Application callback routines are called from within the IP's helper thread.

Parameters

socket_ptr	Pointer to the TCP server socket control block.
wait_option	Defines how the service behaves while the connection is being established. The wait options are defined as follows: NX_NO_WAIT (0x00000000) NX_WAIT_FOREVER (0xFFFFFFFF) timeout value (0x00000001 through 0xFFFFFFFE)

Return Values

NX_SUCCESS	(0x00)	Successful TCP server socket accept (passive connect).
NX_NOT_LISTEN_STATE	(0x36)	The server socket supplied is not in a listen state.

NX_IN_PROGRESS	(0x37)	No wait was specified, the connection attempt is in progress.
NX_WAIT_ABORTED	(0x1A)	Requested suspension was aborted by a call to *tx_thread_wait_abort*.
NX_PTR_ERROR	(0x07)	Socket pointer error.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

Yes

Example

```
NX_PACKET_POOL          my_pool;
NX_IP                   my_ip;
NX_TCP_SOCKET           server_socket;

void   port_12_connect_request(NX_TCP_SOCKET *socket_ptr, UINT port)
{

    /* Simply set the semaphore to wakeup the server thread.  */
    tx_semaphore_put(&port_12_semaphore);
}

void  port_12_disconnect_request(NX_TCP_SOCKET *socket_ptr)
{
    /* The client has initiated a disconnect on this socket. This exmaple
       doesn't use this callback.  */
}

void   port_12_server_thread_entry(ULONG id)
{

NX_PACKET   *my_packet;
UINT        status, i;

    /* Assuming that:
         "port_12_semaphore" has already been created with an initial count of 0
         "my_ip" has already been created and the link is enabled
         "my_pool" packet pool has already been created
    */

    /* Create the server socket.  */
    nx_tcp_socket_create(&my_ip, &server_socket, "Port 12 Server Socket",
                            NX_IP_NORMAL, NX_FRAGMENT_OKAY,
```

```
                                NX_IP_TIME_TO_LIVE, 100,
                                NX_NULL, port_12_disconnect_request);

    /* Setup server listening on port 12.  */
    nx_tcp_server_socket_listen(&my_ip, 12, &server_socket, 5,
                                port_12_connect_request);

    /* Loop to process 5 server connections, sending "Hello_and_Goodbye" to
                                each client and then disconnecting.  */
    for (i = 0; i < 5; i++)
    {

        /* Get the semaphore that indicates a client connection request is
           present.  */
        tx_semaphore_get(&port_12_semaphore, TX_WAIT_FOREVER);

        /* Wait for 200 ticks for the client socket connection to complete.  */
        status =  nx_tcp_server_socket_accept(&server_socket, 200);

        /* Check for a successful connection.  */
        if (status == NX_SUCCESS)
        {

            /* Allocate a packet for the "Hello_and_Goodbye" message.  */
            nx_packet_allocate(&my_pool, &my_packet, NX_TCP_PACKET,
                                NX_WAIT_FOREVER);

            /* Place "Hello_and_Goodbye" in the packet.  */
            nx_packet_data_append(my_packet, "Hello_and_Goodbye",
                                sizeof("Hello_and_Goodbye"), &my_pool,
                                NX_WAIT_FOREVER);

            /* Send "Hello_and_Goodbye" to client.  */
            nx_tcp_socket_send(&server_socket, my_packet, 200);

            /* Check for an error.  */
            if (status)
            {

                /* Error, release the packet.  */
                nx_packet_release(my_packet);
            }

             /* Now disconnect the server socket from the client.  */
             nx_tcp_socket_disconnect(&server_socket, 200);
        }

        /* Unaccept the server socket.  Note that unaccept is called even if
           disconnect or accept fails.  */
        nx_tcp_server_socket_unaccept(&server_socket);

        /* Setup server socket for listening with this socket again.  */
        nx_tcp_server_socket_relisten(&my_ip, 12, &server_socket);
    }

    /* We are now done so unlisten on server port 12.  */
    nx_tcp_server_socket_unlisten(&my_ip, 12);

    /* Delete the server socket.  */
    nx_tcp_socket_delete(&server_socket);
}
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_server_socket_listen

Enable listening for client connection on TCP port

Prototype

```
UINT nx_tcp_server_socket_listen(NX_IP *ip_ptr, UINT port,
                                 NX_TCP_SOCKET *socket_ptr,
                                 UINT listen_queue_size,
                                 VOID (*listen_callback)(NX_TCP_SOCKET *socket_ptr,
                                 UINT port));
```

Description

This service enables listening for a client connection request on the specified TCP port. When a client connection request is received, the supplied server socket is bound to the specified port and the supplied listen callback function is called.

The listen callback routine's processing is completely up to the application. It may contain logic to wake up an application thread that subsequently performs an accept operation. If the application already has a thread suspended on accept processing for this socket, the listen callback routine may not be needed.

If the application wishes to handle additional client connections on the same port, the ***nx_tcp_server_socket_relisten*** must be called with an available socket (a socket in the CLOSED state) for the next connection. Until the re-listen service is called, additional client connections are queued. When the maximum queue depth is exceeded, the oldest connection request is dropped in favor of queuing the new connection request. The maximum queue depth is specified by this service.

Application callback routines are called from the internal IP helper thread.

Parameters

ip_ptr	Pointer to previously created IP instance.
port	Port number to listen on (1 through 0xFFFF).
socket_ptr	Pointer to socket to use for the connection.
listen_queue_size	Number of client connection requests that can be queued.
listen_callback	Application function to call when the connection is received. If a NULL is specified, the listen callback feature is disabled.

Return Values

NX_SUCCESS	(0x00)	Successful TCP port listen enable.
NX_MAX_LISTEN	(0x33)	No more listen request structures are available. The constant NX_MAX_LISTEN_REQUESTS in ***nx_api.h*** defines how many active listen requests are possible.
NX_NOT_CLOSED	(0x35)	The supplied server socket is not in a closed state.
NX_ALREADY_BOUND	(0x22)	The supplied server socket is already bound to a port.
NX_DUPLICATE_LISTEN	(0x34)	There is already an active listen request for this port.
NX_INVALID_PORT	(0x46)	Invalid port specified.
NX_PTR_ERROR	(0x07)	Invalid IP or socket pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

No

Example

```
NX_PACKET_POOL          my_pool;
NX_IP                   my_ip;
NX_TCP_SOCKET           server_socket;

void   port_12_connect_request(NX_TCP_SOCKET *socket_ptr, UINT port)
{

    /* Simply set the semaphore to wakeup the server thread.  */
    tx_semaphore_put(&port_12_semaphore);
}

void  port_12_disconnect_request(NX_TCP_SOCKET *socket_ptr)
{
    /* The client has initiated a disconnect on this socket. This exmaple
       doesn't use this callback.  */
}

void   port_12_server_thread_entry(ULONG id)
{

NX_PACKET   *my_packet;
UINT        status, i;

    /* Assuming that:
         "port_12_semaphore" has already been created with an initial count of 0
         "my_ip" has already been created and the link is enabled
         "my_pool" packet pool has already been created
    */

    /* Create the server socket.  */
    nx_tcp_socket_create(&my_ip, &server_socket, "Port 12 Server Socket",
                            NX_IP_NORMAL, NX_FRAGMENT_OKAY,
                            NX_IP_TIME_TO_LIVE, 100,
                            NX_NULL, port_12_disconnect_request);

    /* Setup server listening on port 12.  */
    nx_tcp_server_socket_listen(&my_ip, 12, &server_socket, 5,
                            port_12_connect_request);

    /* Loop to process 5 server connections, sending "Hello_and_Goodbye" to
                            each client and then disconnecting.  */
    for (i = 0; i < 5; i++)
    {

        /* Get the semaphore that indicates a client connection request is present.  */
        tx_semaphore_get(&port_12_semaphore, TX_WAIT_FOREVER);

        /* Wait for 200 ticks for the client socket connection to complete.  */
        status =  nx_tcp_server_socket_accept(&server_socket, 200);

        /* Check for a successful connection.  */
        if (status == NX_SUCCESS)
        {
```

```
            /* Allocate a packet for the "Hello_and_Goodbye" message.  */
            nx_packet_allocate(&my_pool, &my_packet, NX_TCP_PACKET,
                               NX_WAIT_FOREVER);

            /* Place "Hello_and_Goodbye" in the packet.  */
            nx_packet_data_append(my_packet, "Hello_and_Goodbye",
                               sizeof("Hello_and_Goodbye"), &my_pool,
                               NX_WAIT_FOREVER);

            /* Send "Hello_and_Goodbye" to client.  */
            nx_tcp_socket_send(&server_socket, my_packet, 200);

            /* Check for an error.  */
            if (status)
            {

                /* Error, release the packet.  */
                nx_packet_release(my_packet);
            }

             /* Now disconnect the server socket from the client.  */
             nx_tcp_socket_disconnect(&server_socket, 200);
        }

        /* Unaccept the server socket.  Note that unaccept is called even if
           disconnect or accept fails.  */
        nx_tcp_server_socket_unaccept(&server_socket);

        /* Setup server socket for listening with this socket again.  */
        nx_tcp_server_socket_relisten(&my_ip, 12, &server_socket);
    }

    /* We are now done so unlisten on server port 12.  */
    nx_tcp_server_socket_unlisten(&my_ip, 12);

    /* Delete the server socket.  */
    nx_tcp_socket_delete(&server_socket);
}
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get,nx_tcp_server_socket_accept, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_server_socket_relisten

Re-listen for client connection on TCP port

Prototype

```
UINT nx_tcp_server_socket_relisten(NX_IP *ip_ptr, UINT port,
                                   NX_TCP_SOCKET *socket_ptr);
```

Description

This service is called after a connection has been received on a port that was setup previously for listening. The main purpose of this service is to provide a new server socket for the next client connection. If a connection request is queued, the connection will be processed immediately during this service call.

i *The same callback routine specified by the original listen request is also called when a connection is present for this new server socket.*

Parameters

ip_ptr	Pointer to previously created IP instance.
port	Port number to re-listen on (1 through 0xFFFF).
socket_ptr	Socket to use for the next client connection.

Return Values

NX_SUCCESS	(0x00)	Successful TCP port re-listen.
NX_NOT_CLOSED	(0x35)	The supplied server socket is not in a closed state.
NX_ALREADY_BOUND	(0x22)	The supplied server socket is already bound to a port.
NX_INVALID_RELISTEN	(0x47)	There is already a valid socket pointer for this port or the port specified does not have a listen request active.
NX_CONNECTION_PENDING	(0x48)	Same as NX_SUCCESS, except there was a queued connection request and it was processed during this call.

NX_INVALID_PORT	(0x46)	Invalid port specified.
NX_PTR_ERROR	(0x07)	Invalid IP or listen callback pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

No

Example

```
NX_PACKET_POOL          my_pool;
NX_IP                   my_ip;
NX_TCP_SOCKET           server_socket;

void   port_12_connect_request(NX_TCP_SOCKET *socket_ptr, UINT port)
{

    /* Simply set the semaphore to wakeup the server thread.  */
    tx_semaphore_put(&port_12_semaphore);
}

void  port_12_disconnect_request(NX_TCP_SOCKET *socket_ptr)
{
    /* The client has initiated a disconnect on this socket. This exmaple
       doesn't use this callback.  */
}

void   port_12_server_thread_entry(ULONG id)
{

NX_PACKET   *my_packet;
UINT        status, i;

    /* Assuming that:
        "port_12_semaphore" has already been created with an initial count of 0
       "my_ip" has already been created and the link is enabled
       "my_pool" packet pool has already been created
    */

    /* Create the server socket.  */
    nx_tcp_socket_create(&my_ip, &server_socket, "Port 12 Server Socket",
                            NX_IP_NORMAL, NX_FRAGMENT_OKAY,
                            NX_IP_TIME_TO_LIVE, 100,
                            NX_NULL, port_12_disconnect_request);

    /* Setup server listening on port 12.  */
    nx_tcp_server_socket_listen(&my_ip, 12, &server_socket, 5,
                            port_12_connect_request);

    /* Loop to process 5 server connections, sending "Hello_and_Goodbye" to
                            each client and then disconnecting.  */
```

```
    for (i = 0; i < 5; i++)
    {

        /* Get the semaphore that indicates a client connection request is present.  */
        tx_semaphore_get(&port_12_semaphore, TX_WAIT_FOREVER);

        /* Wait for 200 ticks for the client socket connection to complete.  */
        status =  nx_tcp_server_socket_accept(&server_socket, 200);

        /* Check for a successful connection.  */
        if (status == NX_SUCCESS)
        {

            /* Allocate a packet for the "Hello_and_Goodbye" message.  */
            nx_packet_allocate(&my_pool, &my_packet, NX_TCP_PACKET,
                               NX_WAIT_FOREVER);

            /* Place "Hello_and_Goodbye" in the packet.  */
            nx_packet_data_append(my_packet, "Hello_and_Goodbye",
                               sizeof("Hello_and_Goodbye"), &my_pool,
                               NX_WAIT_FOREVER);

            /* Send "Hello_and_Goodbye" to client.  */
            nx_tcp_socket_send(&server_socket, my_packet, 200);

            /* Check for an error.  */
            if (status)
            {

                /* Error, release the packet.  */
                nx_packet_release(my_packet);
            }

             /* Now disconnect the server socket from the client.  */
             nx_tcp_socket_disconnect(&server_socket, 200);
        }

        /* Unaccept the server socket.  Note that unaccept is called even if
           disconnect or accept fails.  */
        nx_tcp_server_socket_unaccept(&server_socket);

        /* Setup server socket for listening with this socket again.  */
        nx_tcp_server_socket_relisten(&my_ip, 12, &server_socket);
    }

    /* We are now done so unlisten on server port 12.  */
    nx_tcp_server_socket_unlisten(&my_ip, 12);

    /* Delete the server socket.  */
    nx_tcp_socket_delete(&server_socket);
}
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_server_socket_unaccept

Unaccept previous server socket connection

Prototype

```
UINT nx_tcp_server_socket_unaccept(NX_TCP_SOCKET *socket_ptr);
```

Description

This service removes the association between this server socket and the specified server port. The application must call this service after a disconnection or after an unsuccessful accept call.

Parameters

socket_ptr	Pointer to previously setup server socket instance.

Return Values

NX_SUCCESS	(0x00)	Successful server socket unaccept.
NX_NOT_LISTEN_STATE	(0x36)	Server socket is in an improper state, and is probably not disconnected.
NX_PTR_ERROR	(0x07)	Invalid socket pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

No

Example

```
NX_PACKET_POOL          my_pool;
NX_IP                   my_ip;
NX_TCP_SOCKET           server_socket;

void   port_12_connect_request(NX_TCP_SOCKET *socket_ptr, UINT port)
{

    /* Simply set the semaphore to wakeup the server thread.  */
    tx_semaphore_put(&port_12_semaphore);
}

void  port_12_disconnect_request(NX_TCP_SOCKET *socket_ptr)
{
    /* The client has initiated a disconnect on this socket. This exmaple
       doesn't use this callback.  */
}

void   port_12_server_thread_entry(ULONG id)
{

NX_PACKET   *my_packet;
UINT        status, i;

    /* Assuming that:
         "port_12_semaphore" has already been created with an initial count of 0
         "my_ip" has already been created and the link is enabled
         "my_pool" packet pool has already been created
    */

    /* Create the server socket.  */
    nx_tcp_socket_create(&my_ip, &server_socket, "Port 12 Server Socket",
                            NX_IP_NORMAL, NX_FRAGMENT_OKAY,
                            NX_IP_TIME_TO_LIVE, 100,
                            NX_NULL, port_12_disconnect_request);

    /* Setup server listening on port 12.  */
    nx_tcp_server_socket_listen(&my_ip, 12, &server_socket, 5,
                                port_12_connect_request);

    /* Loop to process 5 server connections, sending "Hello_and_Goodbye" to
       each client and then disconnecting.  */
    for (i = 0; i < 5; i++)
    {

        /* Get the semaphore that indicates a client connection request is
           present.  */
        tx_semaphore_get(&port_12_semaphore, TX_WAIT_FOREVER);

        /* Wait for 200 ticks for the client socket connection to complete.  */
        status =  nx_tcp_server_socket_accept(&server_socket, 200);

        /* Check for a successful connection.  */
        if (status == NX_SUCCESS)
        {

            /* Allocate a packet for the "Hello_and_Goodbye" message.  */
            nx_packet_allocate(&my_pool, &my_packet, NX_TCP_PACKET,
                               NX_WAIT_FOREVER);
```

```
            /* Place "Hello_and_Goodbye" in the packet.  */
            nx_packet_data_append(my_packet, "Hello_and_Goodbye",
                                  sizeof("Hello_and_Goodbye"), &my_pool,
                                  NX_WAIT_FOREVER);

            /* Send "Hello_and_Goodbye" to client.  */
            nx_tcp_socket_send(&server_socket, my_packet, 200);

            /* Check for an error.  */
            if (status)
            {

                /* Error, release the packet.  */
                nx_packet_release(my_packet);
            }

             /* Now disconnect the server socket from the client.  */
             nx_tcp_socket_disconnect(&server_socket, 200);
        }

        /* Unaccept the server socket.  Note that unaccept is called even if
           disconnect or accept fails.  */
        nx_tcp_server_socket_unaccept(&server_socket);

        /* Setup server socket for listening with this socket again.  */
        nx_tcp_server_socket_relisten(&my_ip, 12, &server_socket);
    }

    /* We are now done so unlisten on server port 12.  */
    nx_tcp_server_socket_unlisten(&my_ip, 12);

    /* Delete the server socket.  */
    nx_tcp_socket_delete(&server_socket);
}
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get,nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

N E T X

nx_tcp_server_socket_unlisten

Disable listening for client connection on TCP port

Prototype

```
UINT nx_tcp_server_socket_unlisten(NX_IP *ip_ptr, UINT port);
```

Description

This service disables listening for a client connection request on the specified TCP port.

Parameters

ip_ptr	Pointer to previously created IP instance.
port	Number of port to disable listening (0 through 0xFFFF).

Return Values

NX_SUCCESS	(0x00)	Successful TCP listen disable.
NX_ENTRY_NOT_FOUND	(0x16)	Listening was not enabled for thespecified port.
NX_INVALID_PORT	(0x46)	Invalid port specified.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

No

Example

```
NX_PACKET_POOL          my_pool;
NX_IP                   my_ip;
NX_TCP_SOCKET           server_socket;

void   port_12_connect_request(NX_TCP_SOCKET *socket_ptr, UINT port)
{

    /* Simply set the semaphore to wakeup the server thread.  */
    tx_semaphore_put(&port_12_semaphore);
}

void  port_12_disconnect_request(NX_TCP_SOCKET *socket_ptr)
{
    /* The client has initiated a disconnect on this socket. This exmaple
       doesn't use this callback.  */
}

void   port_12_server_thread_entry(ULONG id)
{

NX_PACKET   *my_packet;
UINT        status, i;

    /* Assuming that:
        "port_12_semaphore" has already been created with an initial count of 0
        "my_ip" has already been created and the link is enabled
        "my_pool" packet pool has already been created
    */

    /* Create the server socket.  */
    nx_tcp_socket_create(&my_ip, &server_socket, "Port 12 Server Socket",
                            NX_IP_NORMAL, NX_FRAGMENT_OKAY,
                            NX_IP_TIME_TO_LIVE, 100,
                            NX_NULL, port_12_disconnect_request);

    /* Setup server listening on port 12.  */
    nx_tcp_server_socket_listen(&my_ip, 12, &server_socket, 5,
                                port_12_connect_request);

    /* Loop to process 5 server connections, sending "Hello_and_Goodbye" to
       each client and then disconnecting.  */
    for (i = 0; i < 5; i++)
    {

        /* Get the semaphore that indicates a client connection request is
           present.  */
        tx_semaphore_get(&port_12_semaphore, TX_WAIT_FOREVER);

        /* Wait for 200 ticks for the client socket connection to complete.  */
        status =  nx_tcp_server_socket_accept(&server_socket, 200);

        /* Check for a successful connection.  */
        if (status == NX_SUCCESS)
        {

            /* Allocate a packet for the "Hello_and_Goodbye" message.  */
            nx_packet_allocate(&my_pool, &my_packet, NX_TCP_PACKET,
                               NX_WAIT_FOREVER);
```

```
            /* Place "Hello_and_Goodbye" in the packet.  */
            nx_packet_data_append(my_packet, "Hello_and_Goodbye",
                                  sizeof("Hello_and_Goodbye"), &my_pool,
                                  NX_WAIT_FOREVER);

            /* Send "Hello_and_Goodbye" to client.  */
            nx_tcp_socket_send(&server_socket, my_packet, 200);

            /* Check for an error.  */
            if (status)
            {

                /* Error, release the packet.  */
                nx_packet_release(my_packet);
            }

             /* Now disconnect the server socket from the client.  */
             nx_tcp_socket_disconnect(&server_socket, 200);
        }

        /* Unaccept the server socket.  Note that unaccept is called even if
           disconnect or accept fails.  */
        nx_tcp_server_socket_unaccept(&server_socket);

        /* Setup server socket for listening with this socket again.  */
        nx_tcp_server_socket_relisten(&my_ip, 12, &server_socket);
    }

    /* We are now done so unlisten on server port 12.  */
    nx_tcp_server_socket_unlisten(&my_ip, 12);

    /* Delete the server socket.  */
    nx_tcp_socket_delete(&server_socket);
}
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

N E T X

nx_tcp_socket_bytes_available

Retrieves number of bytes available for retrieval

Prototype

```
UINT nx_tcp_socket_bytes_available(NX_TCP_SOCKET *socket_ptr,
                                   ULONG *bytes_available);
```

Description

This retrieves number of bytes available for retrieval in the specified TCP socket. Note that the TCP socket must already be connected.

Parameters

socket_ptr	Pointer to previously created and connected TCP socket.
bytes_available	Pointer to destination for bytes available.

Return Values

NX_SUCCESS	(0x00)	Service executes successfully. Number of bytes available for read is returned to the caller.
NX_NOT_CONNECTED	(0x38)	Socket is not in a connected state.
NX_PTR_ERROR	(0x07)	Invalid pointers.
NX_NOT_ENABLED	(0x14)	TCP is not enabled.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Threads

Preemption Possible

No

Example

```
/* Get the bytes available for retrieval on the specified socket. */
status = nx_tcp_socket_bytes_available(&my_socket,&bytes_available);

/* Is status = NX_SUCCESS, the available bytes is returned in
   bytes_available. */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get,nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_socket_create

Create TCP client or server socket

Prototype

```
UINT nx_tcp_socket_create(NX_IP *ip_ptr,
                          NX_TCP_SOCKET *socket_ptr, CHAR *name,
                          ULONG type_of_service, ULONG fragment,
                          UINT time_to_live,
                          ULONG window_size,
                          VOID (*urgent_data_callback)(NX_TCP_SOCKET
                          *socket_ptr),
                          VOID (*disconnect_callback)(NX_TCP_SOCKET
                          *socket_ptr));
```

Description

This service creates a TCP client or server socket for the specified IP instance.

i *Application callback routines are called from the thread associated with this IP instance.*

Parameters

ip_ptr	Pointer to previously created IP instance.
socket_ptr	Pointer to new TCP client socket control block.
name	Application name for this TCP socket.
type_of_service	Defines the type of service for the transmission, legal values are as follows: NX_IP_NORMAL (0x00000000) NX_IP_MIN_DELAY (0x00100000) NX_IP_MAX_DATA (0x00080000) NX_IP_MAX_RELIABLE (0x00040000) NX_IP_MIN_COST (0x00020000)
fragment	Specifies whether or not IP fragmenting is allowed. If NX_FRAGMENT_OKAY (0x0) is specified, IP fragmenting is allowed. If NX_DONT_FRAGMENT (0x4000) is specified, IP fragmenting is disabled.

time_to_live	Specifies the 8-bit value that defines how many routers this packet can pass before being thrown away. The default value is specified by NX_IP_TIME_TO_LIVE.
window_size	Defines the maximum number of bytes allowed in the receive queue for this socket
urgent_data_callback	Application function that is called whenever urgent data is detected in the receive stream. If this value is NX_NULL, urgent data is ignored.
disconnect_callback	Application function that is called whenever a disconnect is issued by the socket at the other end of the connection. If this value is NX_NULL, the disconnect callback function is disabled.

Return Values

NX_SUCCESS	(0x00)	Successful TCP client socket create.
NX_OPTION_ERROR	(0x0A)	Invalid type-of-service, fragment, or time-to-live option.
NX_PTR_ERROR	(0x07)	Invalid IP or socket pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Initialization and Threads

Preemption Possible

No

Example

```
/* Create a TCP client socket on the previously created IP instance,
   with normal delivery, IP fragmentation enabled, 0x80 time to
   live, a 200-byte receive window, no urgent callback routine, and
   the "client_disconnect" routine to handle disconnection initiated
   from the other end of the connection.  */
status =  nx_tcp_socket_create(&ip_0, &client_socket,
                      "Client Socket",
                      NX_IP_NORMAL, NX_FRAGMENT_OKAY,
                      0x80, 200, NX_NULL, client_disconnect);

/* If status is NX_SUCCESS, the client socket is created and ready
   to be bound.  */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

N E T X

nx_tcp_socket_delete

Delete TCP socket

Prototype

```
UINT nx_tcp_socket_delete(NX_TCP_SOCKET *socket_ptr);
```

Description

This service deletes a previously created TCP socket.

Parameters

socket_ptr	Previously created TCP socket

Return Values

NX_SUCCESS	(0x00)	Successful socket delete.
NX_NOT_CREATED	(0x27)	Socket was not created.
NX_STILL_BOUND	(0x42)	Socket is still bound.
NX_PTR_ERROR	(0x07)	Invalid socket pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

No

Example

```
/* Delete a previously created TCP client socket.  */
status =  nx_tcp_socket_delete(&client_socket);

/* If status is NX_SUCCESS, the client socket is deleted.  */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_socket_disconnect

Disconnect client and server socket connections

Prototype

```
UINT nx_tcp_socket_disconnect(NX_TCP_SOCKET *socket_ptr,
                              ULONG wait_option);
```

Description

This service disconnects an established client or server socket connection. A disconnect of a server socket should be followed by an un-accept request, while a client socket that is disconnected is left in a state ready for another connection request.

Parameters

socket_ptr	Pointer to previously connected client or server socket instance.
wait_option	Defines how the service behaves while the disconnection is in progress. The wait options are defined as follows: NX_NO_WAIT (0x00000000) NX_WAIT_FOREVER (0xFFFFFFFF) timeout value (0x00000001 through 0xFFFFFFFE)

Return Values

NX_SUCCESS	(0x00)	Successful socket disconnect.
NX_NO_PACKET	(0x01)	No packet available for disconnect message.
NX_NOT_CONNECTED	(0x38)	Specified socket is not connected.
NX_IN_PROGRESS	(0x37)	Disconnect is in progress, no wait was specified.
NX_WAIT_ABORTED	(0x1A)	Requested suspension was aborted by a call to *tx_thread_wait_abort*.
NX_PTR_ERROR	(0x07)	Invalid socket pointer.

NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

Yes

Example

```
/* Disconnect from a previously established connection and wait a
   maximum of 400 timer ticks.  */
status =  nx_tcp_socket_disconnect(&client_socket, 400);

/* If status is NX_SUCCESS, the previously connected socket (either
   as a result of the client socket connect or the server accept) is
   disconnected.  */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get,nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_socket_info_get

Retrieve information about TCP socket activities

Prototype

```
UINT nx_tcp_socket_info_get(NX_TCP_SOCKET *socket_ptr,
                            ULONG *tcp_packets_sent,
                            ULONG *tcp_bytes_sent,
                            ULONG *tcp_packets_received,
                            ULONG *tcp_bytes_received,
                            ULONG *tcp_retransmit_packets,
                            ULONG *tcp_packets_queued,
                            ULONG *tcp_checksum_errors,
                            ULONG *tcp_socket_state,
                            ULONG *tcp_transmit_queue_depth,
                            ULONG *tcp_transmit_window,
                            ULONG *tcp_receive_window);
```

Description

This service retrieves information about TCP socket activities for the specified TCP socket instance.

If a destination pointer is NX_NULL, that particular information is not returned to the caller.

Parameters

socket_ptr	Pointer to previously created TCP socket instance.
tcp_packets_sent	Pointer to destination for the total number of TCP packets sent on socket.
tcp_bytes_sent	Pointer to destination for the total number of TCP bytes sent on socket.
tcp_packets_received	Pointer to destination of the total number of TCP packets received on socket.
tcp_bytes_received	Pointer to destination of the total number of TCP bytes received on socket.
tcp_retransmit_packets	Pointer to destination of the total number of TCP packet retransmissions.

tcp_packets_queued	Pointer to destination of the total number of queued TCP packets on socket.
tcp_checksum_errors	Pointer to destination of the total number of TCP packets with checksum errors on socket.
tcp_socket_state	Pointer to destination of the socket's current state.
tcp_transmit_queue_depth	Pointer to destination of the total number of transmit packets still queued waiting for ACK.
tcp_transmit_window	Pointer to destination of the current transmit window size.
tcp_receive_window	Pointer to destination of the current receive window size.

Return Values

NX_SUCCESS	(0x00)	Successful TCP socket information retrieval.
NX_PTR_ERROR	(0x07)	Invalid socket pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Initialization, threads, and timers

Preemption Possible

No

Example

```
/* Retrieve TCP socket information from previously created socket 0.  */
status =  nx_tcp_socket_info_get(&socket_0,
                                 &tcp_packets_sent,
                                 &tcp_bytes_sent,
                                 &tcp_packets_received,
                                 &tcp_bytes_received,
                                 &tcp_retransmit_packets,
                                 &tcp_packets_queued,
                                 &tcp_checksum_errors,
                                 &tcp_socket_state,
                                 &tcp_transmit_queue_depth,
                                 &tcp_transmit_window,
                                 &tcp_receive_window);

/* If status is NX_SUCCESS, TCP socket information was retrieved.  */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

N E T X

nx_tcp_socket_mss_get

Get MSS of socket

Prototype

```
UINT  nx_tcp_socket_mss_get(NX_TCP_SOCKET *socket_ptr, ULONG *mss);
```

Description

This service retrieves the specified socket's current Maximum Segment Size (MSS).

Parameters

socket_ptr	Pointer to previously created socket.
mss	Destination for returning MSS.

Return Values

NX_SUCCESS	(0x00)	Successful MSS get.
NX_PTR_ERROR	(0x07)	Invalid socket or MSS destination pointer.
NX_NOT_ENABLED	(0x14)	TCP is not enabled.
NX_CALLER_ERROR	(0x11)	Caller is not a thread or initialization.

Allowed From

Initialization and threads

Example

```
/* Get the MSS for the socket "my_socket". */
status =  nx_tcp_socket_mss_get(&my_socket, &mss_value);

/* If status is NX_SUCCESS, the "mss_value" variable contains the
   socket's current MSS value.  */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_receive, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_socket_mss_peer_get

Get MSS of socket peer

Prototype

```
UINT  nx_tcp_socket_mss_peer_get(NX_TCP_SOCKET *socket_ptr, ULONG *mss);
```

Description

This service retrieves the specified socket connected peer's advertised Maximum Segment Size (MSS).

Parameters

socket_ptr	Pointer to previously created and connected socket.
mss	Destination for returning the MSS.

Return Values

NX_SUCCESS	(0x00)	Successful peer MSS get.
NX_PTR_ERROR	(0x07)	Invalid socket or MSS destination pointer.
NX_NOT_ENABLED	(0x14)	TCP is not enabled.
NX_CALLER_ERROR	(0x11)	Caller is not a thread or initialization.

Allowed From

Initialization and threads

Example

```
/* Get the MSS of the connected peer to the socket "my_socket". */
status =  nx_tcp_socket_mss_peer_get(&my_socket, &mss_value);

/* If status is NX_SUCCESS, the "mss_value" variable contains the
   socket peer's advertised MSS value.  */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_set, nx_tcp_socket_receive, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_socket_mss_set

Set MSS of socket

Prototype

```
UINT  nx_tcp_socket_mss_set(NX_TCP_SOCKET *socket_ptr, ULONG mss);
```

Description

This service sets the specified socket's Maximum Segment Size (MSS).

Parameters

socket_ptr	Pointer to previously created socket.
mss	Value of MSS to set.

Return Values

NX_SUCCESS	(0x00)	Successful MSS set.
NX_SIZE_ERROR	(0x09)	Specified MSS value is too large.
NX_PTR_ERROR	(0x07)	Invalid socket pointer.
NX_NOT_ENABLED	(0x14)	TCP is not enabled.
NX_CALLER_ERROR	(0x11)	Caller is not a thread or initialization.

Allowed From

Initialization and threads

Example

```
/* Set the MSS of the socket "my_socket" to 1000 bytes. */
status =  nx_tcp_socket_mss_set(&my_socket, 1000);

/* If status is NX_SUCCESS, the MSS of "my_socket" is 1000 bytes. */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_socket_peer_info_get

Retrieve information about peer TCP socket

Prototype

```
UINT nx_tcp_socket_peer_info_get(NX_TCP_SOCKET *socket_ptr,
                                 ULONG *peer_ip_address, ULONG *peer_port);
```

Description

This service retrieves IP address and port number of the peer socket for a connection.

Parameters

socket_ptr	Pointer to previously created TCP socket.
peer_ip_address	Pointer to destination for peer IP address, in host byte order.
peer_port	Pointer to destination for peer port number, in host byte order.

Return Values

NX_SUCCESS	(0x00)	Service executes successfully. Peer IP address and port number are returned to the caller.
NX_NOT_CONNECTED	(0x38)	Socket is not in a connected state.
NX_PTR_ERROR	(0x07)	Invalid pointers.
NX_NOT_ENABLED	(0x14)	TCP is not enabled.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Threads

Preemption Possible

Yes

Example

```
/* Obtain peer IP address and port on the specified TCP socket. */
status = nx_tcp_socket_peer_info_get(&my_socket, &peer_ip_address, &peer_port);

/* If status = NX_SUCCESS, the data was successfully obtained. */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get,nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_socket_receive

Receive data from TCP socket

Prototype

```
UINT nx_tcp_socket_receive(NX_TCP_SOCKET *socket_ptr,
                           NX_PACKET **packet_ptr, ULONG wait_option);
```

Description

This service receives TCP data from the specified socket. If no data is queued on the specified socket, the caller suspends based on the supplied wait option.

If NX_SUCCESS is returned, the application is responsible for releasing the received packet when it is no longer needed.

Parameters

socket_ptr	Pointer to previously created TCP socket instance.
packet_ptr	Pointer to TCP packet pointer.
wait_option	Defines how the service behaves if do data are currently queued on this socket. The wait options are defined as follows: NX_NO_WAIT (0x00000000) NX_WAIT_FOREVER (0xFFFFFFFF) timeout value (0x00000001 through 0xFFFFFFFE)

Return Values

NX_SUCCESS	(0x00)	Successful socket data receive.
NX_NOT_BOUND	(0x24)	Socket is not bound yet.
NX_NO_PACKET	(0x01)	No data received.
NX_WAIT_ABORTED	(0x1A)	Requested suspension was aborted by a call to *tx_thread_wait_abort*.

NX_NOT_CONNECTED	(0x38)	The socket is no longer connected.
NX_PTR_ERROR	(0x07)	Invalid socket or return packet pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

Yes

Example

```
/* Receive a packet from the previously created and connected TCP
   client socket. If no packet is available, wait for 200 timer ticks
   before giving up.  */
status =  nx_tcp_socket_receive(&client_socket, &packet_ptr, 200);

/* If status is NX_SUCCESS, the received packet is pointed to by
   "packet_ptr".  */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_socket_receive_notify

Notify application of received packets

Prototype

```
UINT nx_tcp_socket_receive_notify(NX_TCP_SOCKET
                        *socket_ptr,
                        VOID (*tcp_receive_notify)(NX_TCP_SOCKET
                        *socket_ptr));
```

Description

This service sets the receive notify function pointer to the callback function specified by the application. This callback function is then called whenever one or more packets are received on the socket. If a NX_NULL pointer is supplied, the notify function is disabled.

Parameters

socket_ptr	Pointer to the TCP socket.
tcp_receive_notify	Application callback function pointer that is called when one or more packets are received on the socket.

Return Values

NX_SUCCESS	(0x00)	Successful socket receive notify.
NX_PTR_ERROR	(0x07)	Invalid socket pointer.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
/* Setup a receive packet callback function for the "client_socket"
   socket. */
status = nx_tcp_socket_receive_notify(client_socket,
                  my_receive_notify);

/* If status is NX_SUCCESS, NetX will call the function named
   "my_receive_notify" whenever one or more packets are received for
   "client_socket". */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_socket_send

Send data through a TCP socket

Prototype

```
UINT nx_tcp_socket_send(NX_TCP_SOCKET *socket_ptr,
                        NX_PACKET *packet_ptr,
                        ULONG wait_option);
```

Description

This service sends TCP data through a previously connected TCP socket. If the receiver's last advertised window size is less than this request, the service optionally suspends based on the wait options specified.

Unless an error is returned, the application should not release the packet after this call. Doing so will cause unpredictable results because the network driver will release the packet after transmission.

Parameters

socket_ptr	Pointer to previously connected TCP socket instance.
packet_ptr	TCP data packet pointer.
wait_option	Defines how the service behaves if the request is greater than the window size of the receiver. The wait options are defined as follows: NX_NO_WAIT (0x00000000) NX_WAIT_FOREVER (0xFFFFFFFF) timeout value (0x00000001 through 0xFFFFFFFE)

Return Values

NX_SUCCESS	(0x00)	Successful socket send.
NX_NOT_BOUND	(0x024)	Socket was not bound to any port.
NX_NOT_CONNECTED	(0x38)	Socket is no longer connected.
NX_ALREADY_SUSPENDED	(0x40)	Another thread is already suspended trying to send data

		on this socket. Only one thread is allowed.
NX_WINDOW_OVERFLOW	(0x39)	Request is greater than receiver's advertised window size in bytes.
NX_WAIT_ABORTED	(0x1A)	Requested suspension was aborted by a call to *tx_thread_wait_abort*.
NX_OVERFLOW	(0x03)	Packet append pointer is invalid.
NX_PTR_ERROR	(0x07)	Invalid socket pointer.
NX_INVALID_PACKET	(0x12)	Packet is not allocated.
NX_TX_QUEUE_DEPTH	(0x49)	Maximum transmit queue depth has been reached.
NX_INVALID_PACKET	(0x12)	Packet is not valid.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

Yes

Example

```
/* Send a packet out on the previously created and connected TCP
   client socket. If the receive window on the other side of the
   connection is less than the packet size, wait 200 timer ticks
   before giving up.  */
status =  nx_tcp_socket_send(&client_socket, packet_ptr, 200);

/* If status is NX_SUCCESS, the packet has been sent!  */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

N E T X

nx_tcp_socket_state_wait

Wait for TCP socket to enter specific state

Prototype

```
UINT nx_tcp_socket_state_wait(NX_TCP_SOCKET *socket_ptr,
                              UINT desired_state,
                              ULONG wait_option);
```

Description

This service waits for the socket to enter the desired state.

Parameters

socket_ptr	Pointer to previously connected TCP socket instance.
desired_state	Desired TCP state. Valid TCP socket states are defined as follows: NX_TCP_CLOSED (0x01) NX_TCP_LISTEN_STATE (0x02) NX_TCP_SYN_SENT (0x03) NX_TCP_SYN_RECEIVED (0x04) NX_TCP_ESTABLISHED (0x05) NX_TCP_CLOSE_WAIT (0x06) NX_TCP_FIN_WAIT_1 (0x07) NX_TCP_FIN_WAIT_2 (0x08) NX_TCP_CLOSING (0x09) NX_TCP_TIMED_WAIT (0x0A) NX_TCP_LAST_ACK (0x0B)
wait_option	Defines how the service behaves if the requested state is not present. The wait options are defined as follows: NX_NO_WAIT (0x00000000) timeout value (0x00000001 through 0xFFFFFFFE)

Return Values

NX_SUCCESS	(0x00)	Successful state wait.
NX_PTR_ERROR	(0x07)	Invalid socket pointer.
NX_NOT_SUCCESSFUL	(0x43)	State not present within the specified wait time.

NX_WAIT_ABORTED	(0x1A)	Requested suspension was aborted by a call to *tx_thread_wait_abort*.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.
NX_OPTION_ERROR	(0x0A)	The desired socket state is invalid.

Allowed From

Threads

Preemption Possible

Yes

Example

```
/* Wait 300 timer ticks for the previously created socket to enter
   the established state in the TCP state machine.  */
status =  nx_tcp_socket_state_wait(&client_socket,
                    NX_TCP_ESTABLISHED, 300);

/* If status is NX_SUCCESS, the socket is now in the established
   state!  */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_transmit_configure, nx_tcp_socket_window_update_notify_set

nx_tcp_socket_transmit_configure

Configure socket's transmit parameters

Prototype

```
UINT  nx_tcp_socket_transmit_configure(NX_TCP_SOCKET
                        *socket_ptr, ULONG max_queue_depth,
                        ULONG timeout, ULONG max_retries,
                        ULONG timeout_shift);
```

Description

This service configures various transmit parameters of the specified TCP socket.

Parameters

socket_ptr	Pointer to the TCP socket.
max_queue_depth	Maximum number of packets allowed to be queued for transmission.
timeout	Number of ThreadX timer ticks an ACK is waited for before the packet is sent again.
max_retries	Maximum number of retries allowed.
timeout_shift	Value to shift the timeout for each subsequent retry. A value of 0, results in the same timeout between successive retries. A value of 1, doubles the timeout between retries.

Return Values

NX_SUCCESS	(0x00)	Successful transmit socket configure.
NX_PTR_ERROR	(0x07)	Invalid socket pointer.
NX_OPTION_ERROR	(0x0a)	Invalid queue depth option.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
/* Configure the "client_socket" for a maximum transmit queue depth of 12,
   100 tick timeouts, a maximum of 20 retries, and a timeout double on each
   successive retry. */
status = nx_tcp_socket_transmit_configure(client_socket, 12, 100, 20, 1);

/* If status is NX_SUCCESS, the socket's transmit retry has been configured.
   */
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get, nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_window_update_notify_set

nx_tcp_socket_window_update_notify_set

Notify application of window size updates

Prototype

```
UINT nx_tcp_socket_window_update_notify_set(NX_TCP_SOCKET *socket_ptr,
        VOID (*tcp_window_update_notify)(NX_TCP_SOCKET *socket_ptr))
```

Description

This service installs a socket window update callback routine. This routine is called automatically whenever the specified socket receives a packet indicating an increase in the window size of the remote host.

Parameters

socket_ptr	Pointer to previously created TCP socket.
tcp_window_update_notify	Callback routine to be called when the window size changes. A value of NULL disables the window change update.

Return Values

NX_SUCCESS	(0x00)	Callback routine is installed on the socket.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_PTR_ERROR	(0x07)	Invalid pointers.
NX_NOT_ENABLED	(0x14)	TCP feature is not enabled.

Allowed From

Initialization, threads, timers

Preemption Possible

No

Example

```
/* Set the function pointer to the windows update callback after creating the
   socket. */
status = nx_tcp_socket_window_update_notify_set(&data_socket,
                                                my_windows_update_callback);

/* Define the window callback function in the host application. */
void    my_windows_update_callback(&data_socket)
{

    /* Process update on increase TCP transmit socket window size. */
    return ;
}
```

See Also

nx_tcp_client_socket_bind, nx_tcp_client_socket_connect, nx_tcp_client_socket_port_get, nx_tcp_client_socket_unbind, nx_tcp_enable, nx_tcp_free_port_find, nx_tcp_info_get,nx_tcp_server_socket_accept, nx_tcp_server_socket_listen, nx_tcp_server_socket_relisten, nx_tcp_server_socket_unaccept, nx_tcp_server_socket_unlisten, nx_tcp_socket_bytes_available, nx_tcp_socket_create, nx_tcp_socket_delete, nx_tcp_socket_disconnect, nx_tcp_socket_info_get, nx_tcp_socket_mss_get, nx_tcp_socket_mss_peer_get, nx_tcp_socket_mss_set, nx_tcp_socket_peer_info_get, nx_tcp_socket_receive, nx_tcp_socket_receive_notify, nx_tcp_socket_send, nx_tcp_socket_state_wait, nx_tcp_socket_transmit_configure

nx_udp_enable

Enable UDP component of NetX

Prototype

```
UINT nx_udp_enable(NX_IP *ip_ptr);
```

Description

This service enables the User Datagram Protocol (UDP) component of NetX. After enabled, UDP datagrams may be sent and received by the application.

Parameters

ip_ptr	Pointer to previously created IP instance.

Return Values

NX_SUCCESS	(0x00)	Successful UDP enable.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_ALREADY_ENABLED	(0x15)	This component has already been enabled.

Allowed From

Initialization, threads, timers

Preemption Possible

No

Example

```
/* Enable UDP on the previously created IP instance.  */
status =  nx_udp_enable(&ip_0);

/* If status is NX_SUCCESS, UDP is now enabled on the specified IP
   instance.  */
```

See Also

nx_udp_free_port_find, nx_udp_info_get, nx_udp_packet_info_extract, nx_udp_socket_bind, nx_udp_socket_bytes_available, nx_udp_socket_checksum_disable, nx_udp_socket_checksum_enable, nx_udp_socket_create, nx_udp_socket_delete, nx_udp_socket_info_get, nx_udp_socket_interface_send, nx_udp_socket_port_get, nx_udp_socket_receive, nx_udp_socket_receive_notify, nx_udp_socket_send, nx_udp_socket_unbind, nx_udp_source_extract

nx_udp_free_port_find

Find next available UDP port

Prototype

```
UINT nx_udp_free_port_find(NX_IP *ip_ptr, UINT port,
                           UINT *free_port_ptr);
```

Description

This service starts looking for a free UDP port (unbound) starting from the application supplied port. The search logic will wrap around if the search happens to reach the maximum port value of 0xFFFF. If the search is successful, the free port is returned in the variable pointed to by free_port_ptr.

This service can be called from another thread and can have the same port returned. To prevent this race condition, the application may wish to place this service and the actual socket bind under the protection of a mutex.

Parameters

ip_ptr	Pointer to previously created IP instance.
port	Port number to start search (1 through 0xFFFF).
free_port_ptr	Pointer to the destination free port return variable.

Return Values

NX_SUCCESS	(0x00)	Successful free port find.
NX_NO_FREE_PORTS	(0x45)	No free ports found.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.
NX_INVALID_PORT	(0x46)	Specified port number is invalid.

Allowed From

Threads, timers

Preemption Possible

No

Example

```
/* Locate a free UDP port, starting at port 12, on a previously
   created IP instance.  */
status = nx_udp_free_port_find(&ip_0, 12, &free_port);

/* If status is NX_SUCCESS pointer, "free_port" identifies the next
   free UDP port on the IP instance.  */
```

See Also

nx_udp_enable, nx_udp_info_get, nx_udp_packet_info_extract, nx_udp_socket_bind, nx_udp_socket_bytes_available, nx_udp_socket_checksum_disable, nx_udp_socket_checksum_enable, nx_udp_socket_create, nx_udp_socket_delete, nx_udp_socket_info_get, nx_udp_socket_interface_send, nx_udp_socket_port_get, nx_udp_socket_receive, nx_udp_socket_receive_notify, nx_udp_socket_send, nx_udp_socket_unbind, nx_udp_source_extract

nx_udp_info_get

Retrieve information about UDP activities

Prototype

```
UINT nx_udp_info_get(NX_IP *ip_ptr,
                     ULONG *udp_packets_sent,
                     ULONG *udp_bytes_sent,
                     ULONG *udp_packets_received,
                     ULONG *udp_bytes_received,
                     ULONG *udp_invalid_packets,
                     ULONG *udp_receive_packets_dropped,
                     ULONG *udp_checksum_errors);
```

Description

This service retrieves information about UDP activities for the specified IP instance.

If a destination pointer is NX_NULL, that particular information is not returned to the caller.

Parameters

ip_ptr	Pointer to previously created IP instance.
udp_packets_sent	Pointer to destination for the total number of UDP packets sent.
udp_bytes_sent	Pointer to destination for the total number of UDP bytes sent.
udp_packets_received	Pointer to destination of the total number of UDP packets received.
udp_bytes_received	Pointer to destination of the total number of UDP bytes received.
udp_invalid_packets	Pointer to destination of the total number of invalid UDP packets.
udp_receive_packets_dropped	Pointer to destination of the total number of UDP receive packets dropped.
udp_checksum_errors	Pointer to destination of the total number of UDP packets with checksum errors.

Return Values

NX_SUCCESS	(0x00)	Successful UDP information retrieval.
NX_PTR_ERROR	(0x07)	Invalid IP pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Initialization, threads, and timers

Preemption Possible

No

Example

```
/* Retrieve UDP information from previously created IP Instance 0.  */
status =  nx_udp_info_get(&ip_0, &udp_packets_sent,
                          &udp_bytes_sent,
                          &udp_packets_received,
                          &udp_bytes_received,
                          &udp_invalid_packets,
                          &udp_receive_packets_dropped,
                          &udp_checksum_errors);

/* If status is NX_SUCCESS, UDP information was retrieved.  */
```

See Also

nx_udp_enable, nx_udp_free_port_find, nx_udp_packet_info_extract, nx_udp_socket_bind, nx_udp_socket_bytes_available, nx_udp_socket_checksum_disable, nx_udp_socket_checksum_enable, nx_udp_socket_create, nx_udp_socket_delete, nx_udp_socket_info_get, nx_udp_socket_interface_send, nx_udp_socket_port_get, nx_udp_socket_receive, nx_udp_socket_receive_notify, nx_udp_socket_send, nx_udp_socket_unbind, nx_udp_source_extract

nx_udp_packet_info_extract

Extract network parameters from UDP packet

Prototype

```
UINT nx_udp_packet_info_extract(NX_PACKET *packet_ptr, ULONG *ip_address,
                      UINT *protocol, UINT *port, UINT *interface_index);
```

Description

This function extracts network parameters from a packet received on an incoming interface.

Parameters

packet_ptr	Pointer to packet.
ip_address	Pointer to destination for packet sender IP address.
Protocol	Pointer to destination for packet protocol (UDP).
port	Pointer to destination for packet sender port.
interface_index	Pointer to destination for packet incoming interface index.

Return Values

NX_SUCCESS	(0x00)	Packet interface data successfully extracted.
NX_PTR_ERROR	(0x07)	Invalid pointer input

Allowed From

Initialization, threads, timers, ISRs

Preemption Possible

No

Example

```
/* Extract network data from UDP packet interface.  */
status = nx_udp_packet_info_extract( packet_ptr, &ip_address,
                &protocol, &port, &interface_index)

/* If status is NX_SUCCESS packet data was successfully retrieved. */
```

See Also

nx_udp_enable, nx_udp_free_port_find, nx_udp_info_get, nx_udp_socket_bind, nx_udp_socket_bytes_available, nx_udp_socket_checksum_disable, nx_udp_socket_checksum_enable, nx_udp_socket_create, nx_udp_socket_delete, nx_udp_socket_info_get, nx_udp_socket_interface_send, nx_udp_socket_port_get, nx_udp_socket_receive, nx_udp_socket_receive_notify, nx_udp_socket_send, nx_udp_socket_unbind, nx_udp_source_extract

nx_udp_socket_bind

Bind UDP socket to UDP port

Prototype

```
UINT nx_udp_socket_bind(NX_UDP_SOCKET *socket_ptr, UINT port,
                        ULONG wait_option);
```

Description

This service binds the previously created UDP socket to the specified UDP port. Valid UDP sockets range from 0 through 0xFFFF.

Parameters

socket_ptr	Pointer to previously created UDP socket instance.
port	Port number to bind to (1 through 0xFFFF). If port number is NX_ANY_PORT (0x0000), the IP instance will search for the next free port and use that for the binding.
wait_option	Defines how the service behaves if the port is already bound to another socket. The wait options are defined as follows: NX_NO_WAIT (0x00000000) NX_WAIT_FOREVER (0xFFFFFFFF) timeout value (0x00000001 through 0xFFFFFFFE)

Return Values

NX_SUCCESS	(0x00)	Successful socket bind.
NX_ALREADY_BOUND	(0x22)	This socket is already bound to another port.
NX_PORT_UNAVAILABLE	(0x23)	Port is already bound to a different socket.
NX_NO_FREE_PORTS	(0x45)	No free port.
NX_WAIT_ABORTED	(0x1A)	Requested suspension was aborted by a call to *tx_thread_wait_abort*.

NX_INVALID_PORT	(0x46)	Invalid port specified.
NX_PTR_ERROR	(0x07)	Invalid socket pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

Yes

Example

```
/* Bind the previously created UDP socket to port 12 on the previously
   created IP instance. If the port is already bound, wait for 300
   timer ticks before giving up. */
status = nx_udp_socket_bind(&udp_socket, 12, 300);

/* If status is NX_SUCCESS, the UDP socket is now bound to port 12.  */
```

See Also

nx_udp_enable, nx_udp_free_port_find, nx_udp_info_get, nx_udp_packet_info_extract, nx_udp_socket_bytes_available, nx_udp_socket_checksum_disable, nx_udp_socket_checksum_enable, nx_udp_socket_create,nx_udp_socket_delete, nx_udp_socket_info_get, nx_udp_socket_interface_send, nx_udp_socket_port_get, nx_udp_socket_receive, nx_udp_socket_receive_notify, nx_udp_socket_send, nx_udp_socket_unbind, nx_udp_source_extract

nx_udp_socket_bytes_available

Retrieves number of bytes available for retrieval

Prototype

```
UINT nx_udp_socket_bytes_available(NX_UDP_SOCKET *socket_ptr,
                                   ULONG *bytes_available)
```

Description

This service retrieves number of bytes available for retrieval in the specified UDP socket.

Parameters

socket_ptr	Pointer to previously created UDP socket.
bytes_available	Pointer to destination for bytes available.

Return Values

NX_SUCCESS	(0x00)	Successful bytes available retrieval.
NX_NOT_SUCCESSFUL	(0x43)	Socket not bound to a port.
NX_PTR_ERROR	(0x07)	Invalid pointers.
NX_NOT_ENABLED	(0x14)	UDP feature is not enabled.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.

Allowed From

Threads

Preemption Possible

No

Example

```
/* Get the bytes available for retrieval from the UDP socket. */
status = nx_udp_socket_bytes_available(&my_socket, &bytes_available);

/* If status = NX_SUCCESS, the number of bytes was successfully retrieved.*/
```

See Also

nx_udp_enable, nx_udp_free_port_find, nx_udp_info_get, nx_udp_packet_info_extract, nx_udp_socket_bind, nx_udp_socket_checksum_disable, nx_udp_socket_checksum_enable, nx_udp_socket_create, nx_udp_socket_delete, nx_udp_socket_info_get, nx_udp_socket_interface_send, nx_udp_socket_port_get, nx_udp_socket_receive, nx_udp_socket_receive_notify, nx_udp_socket_send, nx_udp_socket_unbind, nx_udp_source_extract

nx_udp_socket_checksum_disable

Disable checksum for UDP socket

Prototype

```
UINT nx_udp_socket_checksum_disable(NX_UDP_SOCKET *socket_ptr);
```

Description

This service disables the checksum logic for the specified UDP socket. When the checksum logic is disabled, a value of zero is loaded into the UDP header's checksum field for all packets sent through this socket.

Parameters

socket_ptr	Pointer to previously created UDP socket instance.

Return Values

NX_SUCCESS	(0x00)	Successful socket checksum disable.
NX_NOT_BOUND	(0x24)	Socket is not bound.
NX_PTR_ERROR	(0x07)	Invalid socket pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Initialization, threads, timer

Preemption Possible

No

Example

```
/* Disable the UDP checksum logic for packets sent on this socket.  */
status = nx_udp_socket_checksum_disable(&udp_socket);

/* If status is NX_SUCCESS, outgoing packets will not have a checksum
   calculated.  */
```

See Also

nx_udp_enable, nx_udp_free_port_find, nx_udp_info_get, nx_udp_packet_info_extract, nx_udp_socket_bind, nx_udp_socket_bytes_available, nx_udp_socket_checksum_enable, nx_udp_socket_create, nx_udp_socket_delete, nx_udp_socket_info_get, nx_udp_socket_interface_send, nx_udp_socket_port_get, nx_udp_socket_receive, nx_udp_socket_receive_notify, nx_udp_socket_send, nx_udp_socket_unbind, nx_udp_source_extract

nx_udp_socket_checksum_enable

Enable checksum for UDP socket

Prototype

```
UINT nx_udp_socket_checksum_enable(NX_UDP_SOCKET *socket_ptr);
```

Description

This service enables the checksum logic for the specified UDP socket. The checksum covers the entire UDP data area as well as a pseudo IP header.

Parameters

socket_ptr	Pointer to previously created UDP socket instance.

Return Values

NX_SUCCESS	(0x00)	Successful socket checksum enable.
NX_NOT_BOUND	(0x24)	Socket is not bound.
NX_PTR_ERROR	(0x07)	Invalid socket pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Initialization, threads, timer

Preemption Possible

No

Example

```
/* Enable the UDP checksum logic for packets sent on this socket.  */
status = nx_udp_socket_checksum_enable(&udp_socket);

/* If status is NX_SUCCESS, outgoing packets will have a checksum
   calculated.  */
```

See Also

nx_udp_enable, nx_udp_free_port_find, nx_udp_info_get, nx_udp_packet_info_extract, nx_udp_socket_bind, nx_udp_socket_bytes_available, nx_udp_socket_checksum_disable, nx_udp_socket_create, nx_udp_socket_delete, nx_udp_socket_info_get, nx_udp_socket_interface_send, nx_udp_socket_port_get, nx_udp_socket_receive, nx_udp_socket_receive_notify, nx_udp_socket_send, nx_udp_socket_unbind, nx_udp_source_extract

nx_udp_socket_create

Create UDP socket

Prototype

```
UINT nx_udp_socket_create(NX_IP *ip_ptr,
                          NX_UDP_SOCKET *socket_ptr, CHAR *name,
                          ULONG type_of_service, ULONG fragment,
                          UINT time_to_live, ULONG queue_maximum);
```

Description

This service creates a UDP socket for the specified IP instance.

Parameters

ip_ptr	Pointer to previously created IP instance.
socket_ptr	Pointer to new UDP socket control bloc.
name	Application name for this UDP socket.
type_of_service	Defines the type of service for the transmission, legal values are as follows: NX_IP_NORMAL (0x00000000) NX_IP_MIN_DELAY (0x00100000) NX_IP_MAX_DATA (0x00080000) NX_IP_MAX_RELIABLE (0x00040000) NX_IP_MIN_COST (0x00020000)
fragment	Specifies whether or not IP fragmenting is allowed. If NX_FRAGMENT_OKAY (0x0) is specified, IP fragmenting is allowed. If NX_DONT_FRAGMENT (0x4000) is specified, IP fragmenting is disabled.
time_to_live	Specifies the 8-bit value that defines how many routers this packet can pass before being thrown away. The default value is specified by NX_IP_TIME_TO_LIVE.
queue_maximum	Defines the maximum number of UDP datagrams that can be queued for this socket. After the queue limit is reached, for every new packet received the oldest UDP packet is released.

Return Values

NX_SUCCESS	(0x00)	Successful UDP socket create.
NX_OPTION_ERROR	(0x0A)	Invalid type-of-service, fragment, or time-to-live option.
NX_PTR_ERROR	(0x07)	Invalid IP or socket pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Initialization and Threads

Preemption Possible

No

Example

```
/* Create a UDP socket with a maximum receive queue of 30 packets.  */
status = nx_udp_socket_create(&ip_0, &udp_socket, "Sample UDP Socket",
                              NX_IP_NORMAL, NX_FRAGMENT_OKAY, 0x80, 30);

/* If status is NX_SUCCESS, the new UDP socket has been created and is
   ready for binding.  */
```

See Also

nx_udp_enable, nx_udp_free_port_find, nx_udp_info_get, nx_udp_packet_info_extract, nx_udp_socket_bind, nx_udp_socket_bytes_available, nx_udp_socket_checksum_disable, nx_udp_socket_checksum_enable, nx_udp_socket_delete, nx_udp_socket_info_get, nx_udp_socket_interface_send, nx_udp_socket_port_get, nx_udp_socket_receive, nx_udp_socket_receive_notify, nx_udp_socket_send, nx_udp_socket_unbind, nx_udp_source_extract

nx_udp_socket_delete

Delete UDP socket

Prototype

```
UINT nx_udp_socket_delete(NX_UDP_SOCKET *socket_ptr);
```

Description

This service deletes a previously created UDP socket.

Parameters

socket_ptr	Pointer to previously created UDP socket instance.

Return Values

NX_SUCCESS	(0x00)	Successful socket delete.
NX_NOT_CREATED	(0x27)	Socket was not created.
NX_STILL_BOUND	(0x42)	Socket is still bound.
NX_PTR_ERROR	(0x07)	Invalid socket pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

No

Example

```
/* Delete a previously created UDP socket.  */
status = nx_udp_socket_delete(&udp_socket);

/* If status is NX_SUCCESS, the previously created UDP socket has
   been deleted.  */
```

See Also

nx_udp_enable, nx_udp_free_port_find, nx_udp_info_get, nx_udp_packet_info_extract, nx_udp_socket_bind, nx_udp_socket_bytes_available, nx_udp_socket_checksum_disable, nx_udp_socket_checksum_enable, nx_udp_socket_create, nx_udp_socket_info_get, nx_udp_socket_interface_send, nx_udp_socket_port_get, nx_udp_socket_receive, nx_udp_socket_receive_notify, nx_udp_socket_send, nx_udp_socket_unbind, nx_udp_source_extract

nx_udp_socket_info_get

Retrieve information about UDP socket activities

Prototype

```
UINT nx_udp_socket_info_get(NX_UDP_SOCKET *socket_ptr,
                            ULONG *udp_packets_sent,
                            ULONG *udp_bytes_sent,
                            ULONG *udp_packets_received,
                            ULONG *udp_bytes_received,
                            ULONG *udp_packets_queued,
                            ULONG *udp_receive_packets_dropped,
                            ULONG *udp_checksum_errors);
```

Description

This service retrieves information about UDP socket activities for the specified UDP socket instance.

If a destination pointer is NX_NULL, that particular information is not returned to the caller.

Parameters

socket_ptr	Pointer to previously created UDP socket instance.
udp_packets_sent	Pointer to destination for the total number of UDP packets sent on socket.
udp_bytes_sent	Pointer to destination for the total number of UDP bytes sent on socket.
udp_packets_received	Pointer to destination of the total number of UDP packets received on socket.
udp_bytes_received	Pointer to destination of the total number of UDP bytes received on socket.
udp_packets_queued	Pointer to destination of the total number of queued UDP packets on socket.
udp_receive_packets_dropped	Pointer to destination of the total number of UDP receive packets dropped for socket due to queue size being exceeded.

udp_checksum_errors	Pointer to destination of the total number of UDP packets with checksum errors on socket.

Return Values

NX_SUCCESS	(0x00)	Successful UDP socket information retrieval.
NX_PTR_ERROR	(0x07)	Invalid socket pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Initialization, threads, and timers

Preemption Possible

No

Example

```
/* Retrieve UDP socket information from previously created socket 0.  */
status =  nx_udp_socket_info_get(&socket_0, &udp_packets_sent,
                                            &udp_bytes_sent,
                                            &udp_packets_received,
                                            &udp_bytes_received,
                                            &udp_queued_packets,
                                            &udp_receive_packets_dropped,
                                            &udp_checksum_errors);

/* If status is NX_SUCCESS, UDP socket information was retrieved.  */
```

See Also

nx_udp_enable, nx_udp_free_port_find, nx_udp_info_get, nx_udp_packet_info_extract, nx_udp_socket_bind, nx_udp_socket_bytes_available, nx_udp_socket_checksum_disable, nx_udp_socket_checksum_enable, nx_udp_socket_create, nx_udp_socket_delete, nx_udp_info_get, nx_udp_socket_interface_send, nx_udp_socket_port_get, nx_udp_socket_receive, nx_udp_socket_receive_notify, nx_udp_socket_send, nx_udp_socket_unbind, nx_udp_source_extract

nx_udp_socket_interface_send

Send datagram through UDP socket

Prototype

```
UINT nx_udp_socket_interface_send(NX_UDP_SOCKET *socket_ptr,
                        NX_PACKET **packet_ptr, ULONG ip_address, UINT
                        port, UINT interface_index)
```

Description

This function sends a UDP packet through the specified network interface.

Parameters

socket_ptr	Socket to transmit the packet out on.
packet_ptr	Pointer to packet to transmit.
ip_address	Destination IP address to send packet.
Port	Destination port.
interface_index	Index of interface to send packet on.

Return Values

NX_SUCCESS	(0x00)	Packet successfully sent.
NX_NOT_BOUND	(0x24)	Socket not bound to a port.
NX_IP_ADDRESS_ERROR	(0x21)	Invalid IP address.
NX_NOT_ENABLED	(0x14)	UDP processing not enabled.
NX_PTR_ERROR	(0x07)	Invalid pointer.
NX_OVERFLOW	(0x03)	Invalid packet append pointer.
NX_UNDERFLOW	(0x02)	Invalid packet prepend pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_INVALID_INTERFACE	(0x4C)	Invalid interface index.
NX_INVALID_PORT	(0x46)	Port number exceeds maximum port number.

Allowed From

Threads

Preemption Possible

No

Example

```
/* Send packet out on port 80 to the specified destination IP on the
   interface at index 1 in the IP task interface list.  */
status = nx_udp_packet_interface_send(socket_ptr, packet_ptr,
                                      destination_ip, 80, 1);

/* If status is NX_SUCCESS packet was successfully transmitted. */
```

See Also

nx_udp_enable, nx_udp_free_port_find, nx_udp_info_get, nx_udp_packet_info_extract, nx_udp_socket_bind, nx_udp_socket_checksum_disable, nx_udp_socket_checksum_enable, nx_udp_socket_bytes_available, nx_udp_socket_create, nx_udp_socket_delete, nx_udp_socket_info_get, nx_udp_socket_port_get, nx_udp_socket_receive, nx_udp_socket_receive_notify, nx_udp_socket_send, nx_udp_socket_unbind, nx_udp_source_extract

nx_udp_socket_port_get

Pick up port number bound to UDP socket

Prototype

```
UINT nx_udp_socket_port_get(NX_UDP_SOCKET *socket_ptr,
                            UINT *port_ptr);
```

Description

This service retrieves the port number associated with the socket, which is useful to find the port allocated by NetX in situations where the NX_ANY_PORT was specified at the time the socket was bound.

Parameters

socket_ptr	Pointer to previously created UDP socket instance.
port_ptr	Pointer to destination for the return port number. Valid port numbers are (1- 0xFFFF).

Return Values

NX_SUCCESS	(0x00)	Successful socket bind.
NX_NOT_BOUND	(0x24)	This socket is not bound to a port.
NX_PTR_ERROR	(0x07)	Invalid socket pointer or port return pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads, timers

Preemption Possible

No

Example

```
/* Get the port number of previously created and bound UDP socket.  */
status = nx_udp_socket_port_get(&udp_socket, &port);

/* If status is NX_SUCCESS, the port variable contains the port this
   socket is bound to.  */
```

See Also

nx_udp_enable, nx_udp_free_port_find, nx_udp_info_get, nx_udp_packet_info_extract, nx_udp_socket_bind, nx_udp_socket_bytes_available, nx_udp_socket_checksum_disable, nx_udp_socket_checksum_enable, nx_udp_socket_create, nx_udp_socket_delete, nx_udp_socket_info_get, nx_udp_socket_interface_send, nx_udp_socket_receive, nx_udp_socket_receive_notify, nx_udp_socket_send, nx_udp_socket_unbind, nx_udp_source_extract

nx_udp_socket_receive

Receive datagram from UDP socket

Prototype

```
UINT nx_udp_socket_receive(NX_UDP_SOCKET *socket_ptr,
                           NX_PACKET **packet_ptr, ULONG wait_option);
```

Description

This service receives an UDP datagram from the specified socket. If no datagram is queued on the specified socket, the caller suspends based on the supplied wait option.

If NX_SUCCESS is returned, the application is responsible for releasing the received packet when it is no longer needed.

Parameters

socket_ptr	Pointer to previously created UDP socket instance.
packet_ptr	Pointer to UDP datagram packet pointer.
wait_option	Defines how the service behaves if a datagram is not currently queued on this socket. The wait options are defined as follows:

NX_NO_WAIT	(0x00000000)
NX_WAIT_FOREVER	(0xFFFFFFFF)
timeout value	(0x00000001 through 0xFFFFFFFE)

Return Values

NX_SUCCESS	(0x00)	Successful socket receive.
NX_NOT_BOUND	(0x24)	Socket was not bound to any port.
NX_NO_PACKET	(0x01)	There was no UDP datagram to receive.

NX_WAIT_ABORTED	(0x1A)	Requested suspension was aborted by a call to *tx_thread_wait_abort*.
NX_PTR_ERROR	(0x07)	Invalid socket or packet return pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

Yes

Example

```
/* Receive a packet from a previously created and bound UDP socket.
   If no packets are currently available, wait for 500 timer ticks
   before giving up.  */
status = nx_udp_socket_receive(&udp_socket, &packet_ptr, 500);

/* If status is NX_SUCCESS, the received UDP packet is pointed to by
   packet_ptr. */
```

See Also

nx_udp_enable, nx_udp_free_port_find, nx_udp_info_get, nx_udp_packet_info_extract, nx_udp_socket_bind, nx_udp_socket_bytes_available, nx_udp_socket_checksum_disable, nx_udp_socket_checksum_enable, nx_udp_socket_create, nx_udp_socket_delete, nx_udp_socket_info_get, nx_udp_socket_interface_send, nx_udp_socket_port_get, nx_udp_socket_receive_notify, nx_udp_socket_send, nx_udp_socket_unbind, nx_udp_source_extract

nx_udp_socket_receive_notify

Notify application of each received packet

Prototype

```
UINT nx_udp_socket_receive_notify(NX_UDP_SOCKET *socket_ptr,
                          VOID (*udp_receive_notify)
                          (NX_UDP_SOCKET *socket_ptr));
```

Description

This service sets the receive notify function pointer to the callback function specified by the application. This callback function is then called whenever a packet is received on the socket. If a NX_NULL pointer is supplied, the receive notify function is disabled.

Parameters

socket_ptr	Pointer to the UDP socket.
udp_receive_notify	Application callback function pointer that is called when a packet is received on the socket.

Return Values

NX_SUCCESS	(0x00)	Successful socket receive notify.
NX_PTR_ERROR	(0x07)	Invalid socket pointer.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
/* Setup a receive packet callback function for the "udp_socket"
   socket. */
status = nx_udp_socket_receive_notify(udp_socket,
                                      my_receive_notify);

/* If status is NX_SUCCESS, NetX will call the function named
   "my_receive_notify" whenever a packet is received for
   "udp_socket". */
```

See Also

nx_udp_enable, nx_udp_free_port_find, nx_udp_info_get, nx_udp_packet_info_extract, nx_udp_socket_bind, nx_udp_socket_bytes_available, nx_udp_socket_checksum_disable, nx_udp_socket_checksum_enable, nx_udp_socket_create, nx_udp_socket_delete, nx_udp_socket_info_get, nx_udp_socket_interface_send, nx_udp_socket_port_get, nx_udp_socket_receive, nx_udp_socket_send, nx_udp_socket_unbind, nx_udp_socket_extract

nx_udp_socket_send

Send datagram through UDP socket

Prototype

```
UINT nx_udp_socket_send(NX_UDP_SOCKET *socket_ptr,
                        NX_PACKET *packet_ptr,
                        ULONG ip_address, UINT port);
```

Description

This service sends a UDP datagram through a previously created and bound UDP socket. Note that the service returns immediately, regardless of whether or not the UDP datagram was successfully sent.

Unless an error is returned, the application should not release the packet after this call. Doing so will cause unpredictable results because the network driver will release the packet after transmission.

Parameters

socket_ptr	Pointer to previously created UDP socket instance.
packet_ptr	UDP datagram packet pointer.
ip_address	Destination IP address, which can be a specific host IP address, a network broadcast, an internal loopback, or a multicast address.
port	Destination port number, legal values range between 1 and 0xFFFF.

Return Values

NX_SUCCESS	(0x00)	Successful socket send.
NX_NOT_BOUND	(0x24)	Socket was not bound to any port.
NX_IP_ADDRESS_ERROR	(0x21)	Invalid IP address.
NX_UNDERFLOW	(0x02)	Not enough room to prepend the UPD header in the packet structure.
NX_OVERFLOW	(0x03)	Packet append pointer is invalid.

NX_PTR_ERROR	(0x07)	Invalid socket pointer.
NX_INVALID_PACKET	(0x12)	Packet is not valid.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.
NX_INVALID_PORT	(0x46)	Invalid port specified.

Allowed From

Threads

Preemption Possible

No

Example

```
/* Send a packet through a previously created and bound UDP socket
   to port 12 on IP 1.2.3.5.  */
status = nx_udp_socket_send(&udp_socket, packet_ptr,
                            IP_ADDRESS(1,2,3,5), 12);

/* If status is NX_SUCCESS, the UDP packet was sent.  */
```

See Also

nx_udp_enable, nx_udp_free_port_find, nx_udp_info_get, nx_udp_packet_info_extract, nx_udp_socket_bind, nx_udp_socket_bytes_available, nx_udp_socket_checksum_disable, nx_udp_socket_checksum_enable, nx_udp_socket_create, nx_udp_socket_delete, nx_udp_socket_info_get, nx_udp_socket_interface_send, nx_udp_socket_port_get, nx_udp_socket_receive, nx_udp_socket_receive_notify, nx_udp_socket_unbind, nx_udp_source_extract

nx_udp_socket_unbind

Unbind UDP socket from UDP port

Prototype

```
UINT nx_udp_socket_unbind(NX_UDP_SOCKET *socket_ptr);
```

Description

This service releases the binding between the UDP socket and a UDP port. If there are other threads waiting to bind another socket to the unbound port, the first suspended thread is then bound to the newly unbound port.

Parameters

socket_ptr	Pointer to previously created UDP socket instance.

Return Values

NX_SUCCESS	(0x00)	Successful socket unbind.
NX_NOT_BOUND	(0x24)	Socket was not bound to any port.
NX_PTR_ERROR	(0x07)	Invalid socket pointer.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

Yes

Example

```
/* Unbind the previously bound UDP socket.  */
status = nx_udp_socket_unbind(&udp_socket);

/* If status is NX_SUCCESS, the previously bound socket is now
   unbound.  */
```

See Also

nx_udp_enable, nx_udp_free_port_find, nx_udp_info_get, nx_udp_packet_info_extract, nx_udp_socket_bind, nx_udp_socket_bytes_available, nx_udp_socket_checksum_disable, nx_udp_socket_checksum_enable, nx_udp_socket_create, nx_udp_socket_delete, nx_udp_socket_info_get, nx_udp_socket_interface_send, nx_udp_socket_port_get, nx_udp_socket_receive, nx_udp_socket_receive_notify, nx_udp_socket_send, nx_udp_source_extract

nx_udp_source_extract

Extract IP and sending port from UDP datagram

Prototype

```
UINT nx_udp_source_extract(NX_PACKET *packet_ptr,
                           ULONG *ip_address, UINT *port);
```

Description

This service extracts the sender's IP and port number from the IP and UDP headers of the supplied UDP datagram.

Parameters

packet_ptr	UDP datagram packet pointer.
ip_address	Pointer to the return IP address variable.
port	Pointer to the return port variable.

Return Values

NX_SUCCESS	(0x00)	Successful source IP/port extraction.
NX_INVALID_PACKET	(0x12)	The supplied packet is invalid.
NX_PTR_ERROR	(0x07)	Invalid packet or IP or port destination.
NX_CALLER_ERROR	(0x11)	Invalid caller of this service.
NX_NOT_ENABLED	(0x14)	This component has not been enabled.

Allowed From

Threads

Preemption Possible

No

Example

```
/* Extract the IP and port information from the sender of the UPD
   packet.  */
status = nx_udp_source_extract(packet_ptr, &sender_ip_address,
                               &sender_port);

/* If status is NX_SUCCESS, the sending IP and port information has been
   stored in sender_ip_address and sender_port respectively.  */
```

See Also

nx_udp_enable, nx_udp_free_port_find, nx_udp_info_get, nx_udp_packet_info_extract, nx_udp_socket_bind, nx_udp_socket_bytes_available, nx_udp_socket_checksum_disable, nx_udp_socket_checksum_enable, nx_udp_socket_create, nx_udp_socket_delete, nx_udp_socket_info_get, nx_udp_socket_interface_send, nx_udp_socket_port_get, nx_udp_socket_receive, nx_udp_socket_receive_notify, nx_udp_socket_send, nx_udp_socket_unbind

CHAPTER 5

NetX Network Drivers

This chapter contains a description of network drivers for NetX. The information presented is designed to help developers write application-specific network drivers for NetX. The following topics are covered:

- Driver Introduction 344
- Driver Entry 345
- Driver Requests 345
  - Initialize 346
  - Enable Link 347
  - Disable Link 348
  - Packet Send 348
  - Packet Broadcast 349
  - ARP Send 349
  - ARP Response Send 350
  - RARP Send 351
  - Multicast Group Join 351
  - Multicast Group Leave 352
  - Attach Interface 353
  - Get Link Status 353
  - Get Link Speed 354
  - Get Duplex Type 354
  - Get Error Count 355
  - Get Receive Packet Count 355
  - Get Transmit Packet Count 356
  - Get Allocation Errors 356
  - Driver Deferred Processing 357
  - User Commands 357
  - Unimplemented Commands 358
- Driver Output 358
- Driver Input 359
  - Deferred Receive Packet Handling 360
- Example RAM Ethernet Network Driver 361

Driver Introduction

The **NX_IP** structure contains an array of all active network interfaces on the host application. Driver specific information is stored with each individual network interface in the **NX_INTERFACE** structure defined in ***nx_api.h***.

```
typedef struct NX_IP_DRIVER_STRUCT
{
       UINT             nx_ip_driver_command;
       UINT             nx_ip_driver_status;
       ULONG            nx_ip_driver_physical_address_msw;
       ULONG            nx_ip_driver_physical_address_lsw;
       struct NX_PACKET_STRUCT*nx_ip_driver_packet;
       ULONG            *nx_ip_driver_return_ptr;
       struct NX_IP_STRUCT *nx_ip_driver_ptr;
       struct NX_INTERFACE * nx_ip_driver_interface;
} NX_IP_DRIVER;
```

Driver Entry

The driver entry function for the primary interface (or only interface for single interface hosts) is defined in the ***nx_ip_create*** service call. Driver entry functions for secondary interfaces are defined in the ***nx_ip_interface_attach*** service call in a similar manner. The driver entry function has the following format:

```
VOID    my_driver_entry(NX_IP_DRIVER *request);
```

NetX calls the network driver entry function for sending packets and for various control and status operations including initializing and enabling the network interface. NetX issues commands to the network driver by setting the ***nx_ip_driver_command*** field in the **NX_IP_DRIVER** request structure.

Driver Requests

When NetX interfaces with the physical network, for example to send a packet, it knows which interface the packet must go out on. The driver entry function for that interface is stored in the associated **NX_INTERFACE** control block. NetX creates the driver request with a send command and invokes the driver entry function to execute the command.

Because each network driver has a single entry function, NetX makes all requests through the driver request data structure. The ***nx_ip_driver_command*** member of the driver request data structure (**NX_IP_DRIVER**) defines the request. Status information is reported back to the caller in the member ***nx_ip_driver_status***. If this field is **NX_SUCCESS**, the driver request was completed successfully.

NetX serializes all access to the driver. Therefore, the driver does not need to worry about multiple threads asynchronously calling the entry function.

Initialize

Although the actual driver initialization processing is application and hardware specific, it usually consists of data structure and physical hardware initialization.

The information required by NetX from driver initialization is the Maximum Transmission Unit (MTU) and whether or not the physical interface needs logical to physical mapping. The driver should store this information in the ***nx_interface_ip_mtu_size*** and ***nx_interface_mapping_needed*** fields of the associated **NX_INTERFACE** control block, which is pointed to by ***nx_ip_driver_interface*** in the driver request.

After the application calls ***nx_ip_create,*** or in the case of multihome host applications, after the application calls ***nx_ip_create*** and ***nx_ip_interface_attach***, the IP helper thread initializes each physical network interface associated with the IP instance. It sets the driver command to NX_LINK_INITIALIZE in the driver request and sending the request to the network driver.

The following NX_IP_DRIVER members are used for the initialize request:

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_INITIALIZE
nx_ip_driver_ptr	Pointer to IP instance. This should be saved for use during processing for receive packets.
nx_ip_driver_interface	Pointer to the physical network interface.

The driver is actually called from the IP helper thread that was created for the IP instance. Because of this, it may suspend during the initialization request—if the physical media initialization requires it.

Enable Link

Next, the IP helper thread enables each physical network interface associated with the IP instance by setting the driver command to NX_LINK_ENABLE in the driver request and sending the request to the network driver. This happens shortly after the IP helper thread completes the initialization request. Enabling the link may be as simple as setting the ***nx_interface_link_up*** field in the **NX_INTERFACE** structure pointed to by ***nx_ip_driver_interface***. But it may also involve manipulation of the physical hardware. The following NX_IP_DRIVER members are used for the enable link request:

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_ENABLE
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_interface	Pointer to the physical network interface.

Disable Link

This request is made by NetX during the deletion of an IP instance by the ***nx_ip_delete*** service. This service disables each physical network interface on the IP instance. Disabling the link may be as simple as clearing the ***nx_interface_link_up*** field in the **NX_INTERFACE** structure pointed to by ***nx_ip_driver_interface***. But it may also involve manipulation of the physical hardware. The following

NX_IP_DRIVER members are used for the disable link request:

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_DISABLE
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_interface	Pointer to the physical network interface.

Packet Send

This request is made during internal IP send processing, which all NetX protocols use to transmit packets (except for ARP and RARP). The packet send processing places a physical media header on the front of the packet and then calls the driver's output function to transmit the packet. The following NX_IP_DRIVER members are used for the packet send request:

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_PACKET_SEND
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_packet	Pointer to the packet to send
nx_ip_driver_interface	Pointer to the physical network interface.
nx_ip_driver_physical_address_msw	Most significant 32-bits of physical address (only if physical mapping needed)
nx_ip_driver_physical_address_lsw	Least significant 32-bits of physical address (only if physical mapping needed)

Packet Broadcast

This request is almost identical to the send packet request. The only difference is that the physical address fields are not required because the

destination is a broadcast. The following NX_IP_DRIVER members are used for the packet broadcast request:

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_PACKET_BROADCAST
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_packet	Pointer to the packet to send
nx_ip_driver_physical_address_msw	0x0000FFFF (broadcast)
nx_ip_driver_physical_address_lsw	0xFFFFFFFF (broadcast)
nx_ip_driver_interface	Pointer to the physical network interface

This request is not used internally by NetX so implementation is optional.

ARP Send

This request is also similar to the IP packet send request. The only difference is that the Ethernet header specifies an ARP packet instead of an IP packet, and physical address fields are must be set to broadcast address. The following NX_IP_DRIVER members are used for the ARP send request:

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_ARP_SEND
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_packet	Pointer to the packet to send
nx_ip_driver_physical_address_msw	0x0000FFFF (broadcast)
nx_ip_driver_physical_address_lsw	0xFFFFFFFF (broadcast)
nx_ip_driver_interface	Pointer to the physical network interface

If physical mapping is not needed, implementation of this request is not required.

ARP Response Send

This request is almost identical to the ARP send packet request. The only difference is the physical address fields are required. The following NX_IP_DRIVER members are used for the ARP response send request:

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_ARP_RESPONSE_SEND
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_packet	Pointer to the packet to send
nx_ip_driver_physical_address_msw	Most significant 32-bits of physical address
nx_ip_driver_physical_address_lsw	Least significant 32-bits of physical address
nx_ip_driver_interface	Pointer to the physical network interface

If physical mapping is not needed, implementation of this request is not required.

RARP Send

This request is almost identical to the send packet request. The only differences are the type of packet header and the physical address fields are not required because the physical destination is always a broadcast. The following NX_IP_DRIVER members are used for the RARP send request:

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_RARP_SEND
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_packet	Pointer to the packet to send
nx_ip_driver_physical_address_msw	0x0000FFFF (broadcast)
nx_ip_driver_physical_address_lsw	0xFFFFFFFF (broadcast)
nx_ip_driver_interface	Pointer to the physical network interface

If physical mapping is not needed, implementation of this request is optional.

Multicast Group Join

This request is made with the ***nx_igmp_multicast_interface_join*** service with an input parameter specifying the interface to send the multicast packet out on. The ***nx_igmp_multicast_join*** service is still available, for multicast transmissions on the primary interface. The driver takes the supplied multicast group address and sets up the physical media to accept incoming packets from that address. The following NX_IP_DRIVER members are used for the multicast group join request:

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_MULTICAST_JOIN
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_physical_address_msw	Most significant 32-bits of physical multicast address
nx_ip_driver_physical_address_lsw	Least significant 32-bits of physical multicast address
nx_ip_driver_interface	Pointer to the physical network interface

If multicast capabilities are not required, implementation of this request is not required.

*It is recommended the application use the newer interface specific service instead of the older **nx_igmp_multicast_join** service.*

Multicast Group Leave

This request is invoked from the ***nx_igmp_multicast_leave*** service. The driver removes the supplied Ethernet multicast address from the multicast join list for that physical interface.

After a host has left a multicast group, packets on the network with this Ethernet multicast address are no longer received by the physical interface. The following NX_IP_DRIVER members are used for the multicast group leave request:

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_MULTICAST_LEAVE
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_physical_address_msw	Most significant 32-bits of physical multicast address
nx_ip_driver_physical_address_lsw	Least significant 32-bits of physical multicast address
nx_ip_driver_interface	Pointer to the physical network interface

If multicast capabilities are not required, implementation of this request is not required.

Attach Interface

This request is invoked from the ***nx_ip_interface_attach*** service. The specified interface is attached to the host IP instance. The following NX_IP_DRIVER members are used for the attach interface request:

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_INTERFACE_ATTACH
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_interface	Pointer to the physical network interface
nx_ip_driver_status	Completion status. If the driver is not able to attach the specified interface to the IP instance, it will return a non-zero error status.

Get Link Status

The host application can query the primary interface link status using the NetX service **nx_ip_interface_status_check** service for any interface on the host. See Chapter 4 Description of Services for more details on these services.

The the link status is contained in the ***nx_interface_link_up*** field in the **NX_INTERFACE** structure pointed to by ***nx_ip_driver_interface***.The following NX_IP_DRIVER members are used for the link status request:

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_GET_STATUS
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_return_ptr	Pointer to the destination to place the status.
nx_ip_driver_interface	Pointer to the physical network interface

nx_ip_status_check *is still available for checking the status of the primary interface. However, application developers are encouraged to use the interface specific service.*

Get Link Speed

This request is made from within the ***nx_ip_driver_direct_command*** service. The driver stores the link's line speed in the supplied destination. The following NX_IP_DRIVER members are used for the link line speed request:

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_GET_SPEED
nx_ip_driver_ptr	Pointer to IP instance

NX_IP_DRIVER member	Meaning
nx_ip_driver_return_ptr	Pointer to the destination to place the line speed
nx_ip_driver_interface	Pointer to the physical network interface

This request is not used internally by NetX so its implementation is optional.

Get Duplex Type

This request is made from within the ***nx_ip_driver_direct_command*** service. The driver stores the link's duplex type in the supplied destination. The following NX_IP_DRIVER members are used for the duplex type request:

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_GET_DUPLEX_TYPE
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_return_ptr	Pointer to the destination to place the duplex type
nx_ip_driver_interface	Pointer to the physical network interface

This request is not used internally by NetX so its implementation is optional.

Get Error Count

This request is made from within the ***nx_ip_driver_direct_command*** service. The driver stores the link's error count in the supplied destination. The following NX_IP_DRIVER members

are used for the link error count request:

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_GET_ERROR_COUNT
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_return_ptr	Pointer to the destination to place the error count
nx_ip_driver_interface	Pointer to the physical network interface

This request is not used internally by NetX so its implementation is optional.

Get Receive Packet Count

This request is made from within the ***nx_ip_driver_direct_command*** service. The driver stores the link's receive packet count in the supplied destination. The following NX_IP_DRIVER members are used for the link receive packet count request:

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_GET_RX_COUNT
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_return_ptr	Pointer to the destination to place the receive packet count
nx_ip_driver_interface	Pointer to the physical network interface

This request is not used internally by NetX so its implementation is optional.

Get Transmit Packet Count

This request is made from within the ***nx_ip_driver_direct_command*** service. The driver stores the link's transmit packet count in the supplied destination. The following NX_IP_DRIVER members

are used for the link transmit packet count request:

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_GET_TX_COUNT
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_return_ptr	Pointer to the destination to place the transmit packet count
nx_ip_driver_interface	Pointer to the physical network interface

This request is not used internally by NetX so its implementation is optional.

Get Allocation Errors

This request is made from within the ***nx_ip_driver_direct_command*** service. The driver stores the link's allocation error count in the supplied destination. The following NX_IP_DRIVER members are used for the link allocation error count request:

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_GET_ALLOC_ERRORS
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_return_ptr	Pointer to the destination to place the allocation error count
nx_ip_driver_interface	Pointer to the physical network interface

This request is not used internally by NetX so its implementation is optional.

Driver Deferred Processing

This request is made from the IP helper thread in response to the driver calling the ***_nx_ip_driver_deferred_processing*** routine from a

transmit or receive ISR. This allows the driver ISR to defer the packet receive and transmit processing to the IP helper thread and thus reduce the amount to processing in the ISR. The ***nx_interface_additional_link_info*** field in the **NX_INTERFACE** structure pointed to by ***nx_ip_driver_interface*** may be used by the driver to store information about the deferred processing event from the IP helper thread context. The following NX_IP_DRIVER members are used for the deferred processing event:

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_DEFERRED_PROCESSING
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_interface	Pointer to the physical network interface

User Commands

This request is made from within the ***nx_ip_driver_direct_command*** service. The driver processes the application specific user commands. The following NX_IP_DRIVER members are used for the user command request:

NX_IP_DRIVER member	Meaning
nx_ip_driver_command	NX_LINK_USER_COMMAND
nx_ip_driver_ptr	Pointer to IP instance
nx_ip_driver_return_ptr	User defined
nx_ip_driver_interface	Pointer to the physical network interface

This request is not used internally by NetX so its implementation is optional.

Unimplemented Commands

Commands unimplemented by the network driver must have the return status field set to NX_UNHANDLED_COMMAND.

Driver Output

All previously mentioned packet transmit requests require an output function implemented in the driver. Specific transmit logic is hardware specific, but it usually consists of checking for hardware capacity to send the packet immediately. If possible, the packet payload (and additional payloads in the packet chain) are loaded into one or more of the hardware transmit buffers and a send operation is initiated. If the packet won't fit in the available transmit buffers, the packet is queued.

The recommended transmit queue is a singly linked list, having both head and tail pointers. New packets are added to the end of the queue, keeping the oldest packet at the front. The ***nx_packet_queue_next*** field is used as the packet's next link in the queue. The driver defines the head and tail pointers of the transmit queue.

Because this queue is accessed from thread and interrupt portions of the driver, interrupt protection must be placed around the queue manipulations.

Most physical hardware implementations generate an interrupt upon packet transmit completion. When the driver receives such an interrupt, it calls the ***nx_packet_transmit_release*** service to release the packet associated with the transmit complete interrupt back to the available packet pool. Next, the driver examines the transmit queue for additional packets waiting to be sent. As many of the queued transmit packets that fit into the hardware transmit buffer(s) are

de-queued and loaded into the buffers. This is followed by initiation of another send operation.

Driver Input

Upon reception of a received packet interrupt, the network driver retrieves the packet from the physical hardware receive buffers and builds a valid NetX packet. Building a valid NetX packet involves setting up the appropriate length field and chaining together multiple packets if the incoming packet's size was greater than a single packet payload. After properly built, the physical layer header is removed and the receive packet is dispatched to NetX.

NetX assumes that the IP and ARP headers are aligned on a ULONG boundary. The NetX network driver must therefore ensure this alignment. In Ethernet environments this is done by starting the Ethernet header two bytes from the beginning of the packet. When the Ethernet header is removed, the underlying IP or ARP header is ULONG aligned.

There are several receive packet functions available in NetX. If the received packet is an ARP packet, ***_nx_arp_packet_deferred_receive*** is called. If the received packet is an RARP packet, ***_nx_rarp_packet_deferred_receive*** is called. There are several options for handling incoming IP packets. For the fastest handling of IP packets, ***_nx_ip_packet_receive*** is called. This approach has the least overhead, but requires more processing in the driver's receive interrupt service handler (ISR). For minimal ISR processing ***_nx_ip_packet_deferred_receive*** is called.

After the new receive packet is properly built, the physical hardware's receive buffers are setup to receive more data. This might require allocating NetX

packets and placing the payload address in the hardware receive buffer or it may simply amount to changing a setting in the hardware receive buffer. To minimize overrun possibilities, it is important that the hardware's receive buffers have available buffers as soon as possible after a packet is received.

The initial receive buffers are setup during driver initialization.

Deferred Receive Packet Handling

The driver may defer receive packet processing to the NetX IP helper thread. For some applications this may be necessary to minimize ISR processing as well as dropped packets.

To use deferred packet handling, the NetX library must first be compiled with ***NX_DRIVER_DEFERRED_PROCESSING*** defined. This adds the deferred packet logic to the NetX IP helper thread. Next, the driver must register its deferred handling function before any packets can be processed. This routine should be called during driver initialization. For multihome hosts, unless one driver handles all interfaces, each interface driver must register its deferred handling function. The function prototype for registration of the deferred packet handler follows:

```
VOID  _nx_ip_driver_deferred_enable(NX_IP *ip_ptr,
                VOID (*driver_deferred_packet_handler)(NX_IP *ip_ptr,
                                                       NX_PACKET *packet_ptr));
```

The deferred receive function should be called from the driver's receive interrupt processing. The function prototype for the deferred receive function follows:

```
VOID  _nx_ip_driver_deferred_receive(NX_IP *ip_ptr,
                                     NX_PACKET *packet_ptr);
```

The deferred receive function places the receive

packet represented by ***packet_ptr*** on a FIFO (linked list) and notifies the IP helper thread. After executing, the IP helper repetitively calls the deferred handling function to process each deferred packet. The deferred handler processing typically includes removing the packet's physical layer header (usually Ethernet) and dispatching it to one of these NetX receive functions:

_nx_ip_packet_receive
_nx_arp_packet_deferred_receive
_nx_rarp_packet_deferred_receive

Example RAM Ethernet Network Driver

The NetX demonstration system is delivered with a small RAM Ethernet network driver, defined in the file ***nx_ram_network_driver.c***. For single interface hosts creating multiple IP instances, this driver assumes all the IP instances exchange packets on the same network. This driver assumes the IP instances are all on the same network and simply assigns virtual hardware addresses to each IP instance as they are created. This file provides a good example of the basic structure for NetX network drivers and is listed on the following page.

For multihome hosts, this driver assumes each IP instance interface exchanges packets with another IP instance on the same network interface. The RAM driver assigns virtual hardware addresses to each interface as they are created.

```
/**************************************************************************/
/*                                                                        */
/*            Copyright (c) 1996-2009 by Express Logic Inc.               */
/*                                                                        */
/*  This software is copyrighted by and is the sole property of Express   */
/*  Logic, Inc.  All rights, title, ownership, or other interests         */
/*  in the software remain the property of Express Logic, Inc.  This      */
/*  software may only be used in accordance with the corresponding        */
/*  license agreement.  Any unauthorized use, duplication, transmission,  */
/*  distribution, or disclosure of this software is expressly forbidden.  */
/*                                                                        */
/*  This Copyright notice may not be removed or modified without prior    */
/*  written consent of Express Logic, Inc.                                */
/*                                                                        */
/*  Express Logic, Inc. reserves the right to modify this software        */
/*  without notice.                                                       */
/*                                                                        */
/*  Express Logic, Inc.                     info@expresslogic.com         */
/*  11423 West Bernardo Court               http://www.expresslogic.com   */
/*  San Diego, CA  92127                                                  */
/*                                                                        */
/**************************************************************************/


/**************************************************************************/
/**************************************************************************/
/**                                                                       */
/** NetX Component                                                        */
/**                                                                       */
/**   RAM Network (RAM)                                                   */
/**                                                                       */
/**************************************************************************/
/**************************************************************************/


/* Include necessary system files.  */

#include "nx_api.h"


#define NX_LINK_MTU     8096


/* Define Ethernet address format.  This is prepended to the incoming IP
   and ARP/RARP messages.  The frame beginning is 14 bytes, but for speed
   purposes, we are going to assume there are 16 bytes free in front of the
   prepend pointer and that the prepend pointer is 32-bit aligned.

    Byte Offset     Size            Meaning

        0           6           Destination Ethernet Address
        6           6           Source Ethernet Address
        12          2           Ethernet Frame Type, where:

                                        0x0800 -> IP Datagram
                                        0x0806 -> ARP Request/Reply
                                        0x0835 -> RARP request reply

        42          18          Padding on ARP and RARP messages only.  */

#define NX_ETHERNET_IP      0x0800
#define NX_ETHERNET_ARP     0x0806
#define NX_ETHERNET_RARP    0x8035
#define NX_ETHERNET_SIZE    14

/* For the simulated ethernet driver, physical addresses are allocated starting
   at the preset value and then incremented before the next allocation.  */

ULONG   simulated_address_msw =  0x1122;
ULONG   simulated_address_lsw =  0x33445566;
```

```
/* Define driver prototypes.  */

VOID    _nx_ram_network_driver(NX_IP_DRIVER *driver_req_ptr);
void    _nx_ram_network_driver_output(NX_IP *ip_ptr, NX_PACKET *packet_ptr);
void    _nx_ram_network_driver_receive(NX_IP *ip_ptr, NX_PACKET *packet_ptr);


/**************************************************************************/
/*                                                                        */
/*  FUNCTION                                               RELEASE        */
/*                                                                        */
/*    _nx_ram_network_driver                              PORTABLE C      */
/*                                                           5.3          */
/*  AUTHOR                                                                */
/*                                                                        */
/*    William E. Lamie, Express Logic, Inc.                               */
/*                                                                        */
/*  DESCRIPTION                                                           */
/*                                                                        */
/*    This function acts as a virtual network for testing the NetX source */
/*    and driver concepts.                                                */
/*                                                                        */
/*  INPUT                                                                 */
/*                                                                        */
/*    ip_ptr                                Pointer to IP protocol block  */
/*                                                                        */
/*  OUTPUT                                                                */
/*                                                                        */
/*    None                                                                */
/*                                                                        */
/*  CALLS                                                                 */
/*                                                                        */
/*    _nx_ram_network_driver_output         Send physical packet out      */
/*                                                                        */
/*  CALLED BY                                                             */
/*                                                                        */
/*    NetX IP processing                                                  */
/*                                                                        */
/*  RELEASE HISTORY                                                       */
/*                                                                        */
/*    DATE              NAME                      DESCRIPTION             */
/*                                                                        */
/*  12-12-2005     William E. Lamie         Initial Version 5.0           */
/*  08-09-2007     William E. Lamie         Modified comments and         */
/*                                          changed UL to ULONG cast,     */
/*                                          resulting in version 5.1      */
/*  07-04-2009     William E. Lamie         Modified comments, resulting  */
/*                                          in version 5.2                */
/*  12-31-2009     Yuxin Zhou               Modified comments and         */
/*                                          added multihome support,      */
/*                                          resulting in version 5.3      */
/**************************************************************************/
VOID  _nx_ram_network_driver(NX_IP_DRIVER *driver_req_ptr)
{

NX_IP        *ip_ptr;
NX_PACKET    *packet_ptr;
ULONG        *ethernet_frame_ptr;
NX_INTERFACE *interface_ptr;

    /* Setup the IP pointer from the driver request.  */
    ip_ptr =  driver_req_ptr -> nx_ip_driver_ptr;

    /* Default to successful return.  */
    driver_req_ptr -> nx_ip_driver_status =  NX_SUCCESS;

    /* Setup interface pointer.  */
    interface_ptr = driver_req_ptr -> nx_ip_driver_interface;
```

```
    /* Process according to the driver request type in the IP control
       block.  */
    switch (driver_req_ptr -> nx_ip_driver_command)
    {

        case NX_LINK_INITIALIZE:

        case NX_LINK_INTERFACE_ATTACH:
          break;

        {

            /* Process driver initialization.  */
#ifdef NX_DEBUG
            printf("NetX RAM Driver Initialization - %s\n", ip_ptr -> nx_ip_name);
            printf("  IP Address =%08X\n", interface_ptr -> nx_interface_ip_address);
#endif

            /* Setup the link maximum transfer unit. Note that the MTU should
               take into account the physical header needs and alignment
               requirements. For example, we are going to report actual
               MTU less the ethernet header and 2 bytes to keep alignment. */
             interface_ptr -> nx_interface_ip_mtu_size =  (NX_LINK_MTU - NX_ETHERNET_SIZE - 2);

            /* Setup the physical address of this IP instance.  Increment the
               physical address lsw to simulate multiple nodes hanging on the
               ethernet.  */
            interface_ptr -> nx_interface_physical_address_msw =  simulated_address_msw;
            interface_ptr -> nx_interface_physical_address_lsw =  simulated_address_lsw++;

            /* Indicate to the IP software that IP to physical mapping
               is required.  */
            interface_ptr -> nx_interface_address_mapping_needed =  NX_TRUE;

            break;
        }

        case NX_LINK_ENABLE:
        {

            /* Process driver link enable.  */

            /* In the RAM driver, just set the enabled flag.  */
            interface_ptr -> nx_interface_link_up =  NX_TRUE;

#ifdef NX_DEBUG
            printf("NetX RAM Driver Link Enabled - %s\n", ip_ptr -> nx_ip_name);
#endif
            break;
        }

        case NX_LINK_DISABLE:
        {

            /* Process driver link disable.  */

            /* In the RAM driver, just clear the enabled flag.  */
            interface_ptr -> nx_interface_link_up =  NX_FALSE;

#ifdef NX_DEBUG
            printf("NetX RAM Driver Link Disabled - %s\n", ip_ptr -> nx_ip_name);
#endif
            break;
        }

        case NX_LINK_PACKET_SEND:
        {

            /* Process driver send packet.  */
```

```
            /* Place the ethernet frame at the front of the packet.  */
            packet_ptr =  driver_req_ptr -> nx_ip_driver_packet;

            /* Adjust the prepend pointer.  */
            packet_ptr -> nx_packet_prepend_ptr =  packet_ptr ->
                                                    nx_packet_prepend_ptr - NX_ETHERNET_SIZE;

            /* Adjust the packet length.  */
            packet_ptr -> nx_packet_length =  packet_ptr -> nx_packet_length + NX_ETHERNET_SIZE;

            /* Setup the ethernet frame pointer to build the ethernet frame.  Backup another 2
               bytes to get 32-bit word alignment.  */
            ethernet_frame_ptr =  (ULONG_PTR) (packet_ptr -> nx_packet_prepend_ptr - 2);

            /* Build the ethernet frame.  */
            *ethernet_frame_ptr     =  driver_req_ptr -> nx_ip_driver_physical_address_msw;
            *(ethernet_frame_ptr+1) =  driver_req_ptr -> nx_ip_driver_physical_address_lsw;
            *(ethernet_frame_ptr+2) =  (interface_ptr -> nx_interface_physical_address_msw << 16) |
                                       (interface_ptr -> nx_interface_physical_address_lsw >> 16);
            *(ethernet_frame_ptr+3) =  (interface_ptr -> nx_interface_physical_address_lsw << 16) |
            /* Endian swapping if NX_LITTLE_ENDIAN is defined.  */
            NX_CHANGE_ULONG_ENDIAN(*(ethernet_frame_ptr));
            NX_CHANGE_ULONG_ENDIAN(*(ethernet_frame_ptr+1));
            NX_CHANGE_ULONG_ENDIAN(*(ethernet_frame_ptr+2));
            NX_CHANGE_ULONG_ENDIAN(*(ethernet_frame_ptr+3));
#ifdef NX_DEBUG_PACKET
            printf("NetX RAM Driver Packet Send - %s\n", ip_ptr -> nx_ip_name);
#endif
            _nx_ram_network_driver_output(ip_ptr, packet_ptr);
            break;
        }

        case NX_LINK_PACKET_BROADCAST:
        {

            /* Process driver send packet.  */

            /* Place the ethernet frame at the front of the packet.  */
            packet_ptr =  driver_req_ptr -> nx_ip_driver_packet;

            /* Adjust the prepend pointer.  */
            packet_ptr -> nx_packet_prepend_ptr =  packet_ptr ->
                                                    nx_packet_prepend_ptr - NX_ETHERNET_SIZE;

            /* Adjust the packet length.  */
            packet_ptr -> nx_packet_length =  packet_ptr -> nx_packet_length + NX_ETHERNET_SIZE;

            /* Setup the ethernet frame pointer to build the ethernet frame.  Backup another 2
               bytes to get 32-bit word alignment.  */
            ethernet_frame_ptr =  (ULONG_PTR) (packet_ptr -> nx_packet_prepend_ptr - 2);

            /* Build the ethernet frame.  */
            *ethernet_frame_ptr     =  (ULONG) 0xFFFF;          /* Broadcast!  */
            *(ethernet_frame_ptr+1) =  (ULONG) 0xFFFFFFFF;
            *(ethernet_frame_ptr+2) =  (interface_ptr -> nx_interface_physical_address_msw << 16) |
                                       (interface_ptr -> nx_interface_physical_address_lsw >> 16);
            *(ethernet_frame_ptr+3) =  (interface_ptr -> nx_interface_physical_address_lsw << 16) |

            /* Endian swapping if NX_LITTLE_ENDIAN is defined.  */
            NX_CHANGE_ULONG_ENDIAN(*(ethernet_frame_ptr));
            NX_CHANGE_ULONG_ENDIAN(*(ethernet_frame_ptr+1));
            NX_CHANGE_ULONG_ENDIAN(*(ethernet_frame_ptr+2));
            NX_CHANGE_ULONG_ENDIAN(*(ethernet_frame_ptr+3));

#ifdef NX_DEBUG_PACKET
            printf("NetX RAM Driver Packet Broadcast - %s\n", ip_ptr -> nx_ip_name);
#endif
            _nx_ram_network_driver_output(ip_ptr, packet_ptr);
            break;
        }
```

```
        case NX_LINK_ARP_SEND:
        {

            /* Process driver send packet.  */

            /* Place the ethernet frame at the front of the packet.  */
            packet_ptr =  driver_req_ptr -> nx_ip_driver_packet;

            /* Adjust the prepend pointer.  */
            packet_ptr -> nx_packet_prepend_ptr =
                    packet_ptr ->nx_packet_prepend_ptr - NX_ETHERNET_SIZE;

            /* Adjust the packet length.  */
            packet_ptr -> nx_packet_length =  packet_ptr -> nx_packet_length + NX_ETHERNET_SIZE;

            /* Setup the ethernet frame pointer to build the ethernet frame.  Backup another 2
               bytes to get 32-bit word alignment.  */
            ethernet_frame_ptr =  (ULONG_PTR) (packet_ptr -> nx_packet_prepend_ptr - 2);

            /* Build the ethernet frame.  */
            *ethernet_frame_ptr     =  (ULONG) 0xFFFF;          /* Broadcast!  */
            *(ethernet_frame_ptr+1) =  (ULONG) 0xFFFFFFFF;
            *(ethernet_frame_ptr+2) =  (interface_ptr -> nx_interface_physical_address_msw << 16) |
                                       (interface_ptr -> nx_interface_physical_address_lsw >> 16);
            *(ethernet_frame_ptr+3) =  (interface_ptr -> nx_interface_physical_address_lsw << 16) |

            /* Endian swapping if NX_LITTLE_ENDIAN is defined.  */
            NX_CHANGE_ULONG_ENDIAN(*(ethernet_frame_ptr));
            NX_CHANGE_ULONG_ENDIAN(*(ethernet_frame_ptr+1));
            NX_CHANGE_ULONG_ENDIAN(*(ethernet_frame_ptr+2));
            NX_CHANGE_ULONG_ENDIAN(*(ethernet_frame_ptr+3));

#ifdef NX_DEBUG
            printf("NetX RAM Driver ARP Send - %s\n", ip_ptr -> nx_ip_name);
#endif
            _nx_ram_network_driver_output(ip_ptr, packet_ptr);
            break;
        }

        case NX_LINK_ARP_RESPONSE_SEND:
        {

            /* Process driver send packet.  */

            /* Place the ethernet frame at the front of the packet.  */
            packet_ptr =  driver_req_ptr -> nx_ip_driver_packet;

            /* Adjust the prepend pointer.  */
            packet_ptr -> nx_packet_prepend_ptr =  packet_ptr ->
                                                   nx_packet_prepend_ptr - NX_ETHERNET_SIZE;

            /* Adjust the packet length.  */
            packet_ptr -> nx_packet_length =  packet_ptr -> nx_packet_length + NX_ETHERNET_SIZE;

            /* Setup the ethernet frame pointer to build the ethernet frame.  Backup another 2
               bytes to get 32-bit word alignment.  */
            ethernet_frame_ptr =  (ULONG_PTR) (packet_ptr -> nx_packet_prepend_ptr - 2);

            /* Build the ethernet frame.  */

            *ethernet_frame_ptr     =  driver_req_ptr -> nx_ip_driver_physical_address_msw;
            *(ethernet_frame_ptr+1) =  driver_req_ptr -> nx_ip_driver_physical_address_lsw;
            *(ethernet_frame_ptr+2) =  (interface_ptr -> nx_interface_physical_address_msw << 16) |
                                       (interface_ptr -> nx_interface_physical_address_lsw >> 16);
            *(ethernet_frame_ptr+3) =  (interface_ptr -> nx_interface_physical_address_lsw << 16) |


            /* Endian swapping if NX_LITTLE_ENDIAN is defined.  */
            NX_CHANGE_ULONG_ENDIAN(*(ethernet_frame_ptr));
```

```
            NX_CHANGE_ULONG_ENDIAN(*(ethernet_frame_ptr+1));
            NX_CHANGE_ULONG_ENDIAN(*(ethernet_frame_ptr+2));
            NX_CHANGE_ULONG_ENDIAN(*(ethernet_frame_ptr+3));

#ifdef NX_DEBUG
            printf("NetX RAM Driver ARP Response Send - %s\n", ip_ptr -> nx_ip_name);
#endif
            _nx_ram_network_driver_output(ip_ptr, packet_ptr);
            break;
        }

        case NX_LINK_RARP_SEND:
        {

            /* Process driver send packet.  */

            /* Place the ethernet frame at the front of the packet.  */
            packet_ptr =  driver_req_ptr -> nx_ip_driver_packet;

            /* Adjust the prepend pointer.  */
            packet_ptr -> nx_packet_prepend_ptr =  packet_ptr ->
                                                    nx_packet_prepend_ptr - NX_ETHERNET_SIZE;

            /* Adjust the packet length.  */
            packet_ptr -> nx_packet_length =  packet_ptr -> nx_packet_length + NX_ETHERNET_SIZE;

            /* Setup the ethernet frame pointer to build the ethernet frame.  Backup another 2
               bytes to get 32-bit word alignment.  */
            ethernet_frame_ptr =  (ULONG_PTR) (packet_ptr -> nx_packet_prepend_ptr - 2);

            /* Build the ethernet frame.  */
            *ethernet_frame_ptr     =  (ULONG) 0xFFFF;          /* Broadcast!  */
            *(ethernet_frame_ptr+1) =  (ULONG) 0xFFFFFFFF;
            *(ethernet_frame_ptr+2) =  (interface_ptr -> nx_interface_physical_address_msw << 16) |
                                       (interface_ptr -> nx_interface_physical_address_lsw >> 16);
            *(ethernet_frame_ptr+3) =  (interface_ptr -> nx_interface_physical_address_lsw << 16) |

            /* Endian swapping if NX_LITTLE_ENDIAN is defined.  */
            NX_CHANGE_ULONG_ENDIAN(*(ethernet_frame_ptr));
            NX_CHANGE_ULONG_ENDIAN(*(ethernet_frame_ptr+1));
            NX_CHANGE_ULONG_ENDIAN(*(ethernet_frame_ptr+2));
            NX_CHANGE_ULONG_ENDIAN(*(ethernet_frame_ptr+3));

#ifdef NX_DEBUG
            printf("NetX RAM Driver RARP Send - %s\n", ip_ptr -> nx_ip_name);
#endif
            _nx_ram_network_driver_output(ip_ptr, packet_ptr);
            break;
        }

        case NX_LINK_MULTICAST_JOIN:
        {

            /* For real ethernet devices the hardware registers that support IP multicast
               need to be searched for an open entry.  If found, the multicast ethernet
               address contained in the driver request structure
               (nx_ip_driver_physical_address_msw & nx_ip_driver_physical_address_lsw)
               needs to be loaded into ethernet chip.  If no free entries are found,
               an NX_NO_MORE_ENTRIES error should be returned to the caller.  */
            break;
        }


        case NX_LINK_MULTICAST_LEAVE:
        {

            /* For real ethernet devices the hardware registers that support IP multicast
               need to be searched for a matching entry.  If found, the multicast ethernet
               address should be cleared in the hardware so that a new entry may use it
               on the next join operation.  */
```

```
            break;
        }

        case NX_LINK_GET_STATUS:
        {

            /* Return the link status in the supplied return pointer.  */
            *(driver_req_ptr -> nx_ip_driver_return_ptr) =
                    ip_ptr-> nx_ip_interface[0].nx_interface_link_up;
        }

        default:
        {

            /* Invalid driver request.  */

            /* Return the unhandled command status.  */
            driver_req_ptr -> nx_ip_driver_status =  NX_UNHANDLED_COMMAND;

#ifdef NX_DEBUG
            printf("NetX RAM Driver Received invalid request - %s\n", ip_ptr -> nx_ip_name);
#endif
        }
    }
}


/**************************************************************************/
/*                                                                        */
/*  FUNCTION                                               RELEASE        */
/*                                                                        */
/*    _nx_ram_network_driver_output                       PORTABLE C      */
/*                                                           5.3          */
/*  AUTHOR                                                                */
/*                                                                        */
/*    William E. Lamie, Express Logic, Inc.                               */
/*                                                                        */
/*  DESCRIPTION                                                           */
/*                                                                        */
/*    This function simply sends the packet to the IP instance on the     */
/*    created IP list that matches the physical destination specified in  */
/*    the Ethernet packet.  In a real hardware setting, this routine      */
/*    would simply put the packet out on the wire.                        */
/*                                                                        */
/*  INPUT                                                                 */
/*                                                                        */
/*    ip_ptr                              Pointer to IP protocol block    */
/*    packet_ptr                          Packet pointer                  */
/*                                                                        */
/*  OUTPUT                                                                */
/*                                                                        */
/*    None                                                                */
/*                                                                        */
/*  CALLS                                                                 */
/*                                                                        */
/*    nx_packet_copy                      Copy a packet                   */
/*    nx_packet_transmit_release          Release a packet                */
/*    _nx_ram_network_driver_receive      RAM driver receive processing   */
/*                                                                        */
/*  CALLED BY                                                             */
/*                                                                        */
/*    NetX IP processing                                                  */
/*                                                                        */
/*  RELEASE HISTORY                                                       */
/*                                                                        */
/*    DATE              NAME                      DESCRIPTION             */
/*                                                                        */
/*  12-12-2005     William E. Lamie         Initial Version 5.0           */
/*  08-09-2007     William E. Lamie         Modified comments and         */
```

```
/*                                          changed UL to ULONG cast,     */
/*                                          resulting in version 5.1      */
/*  07-04-2009     William E. Lamie         Modified comments, resulting  */
/*                                          in version 5.2                */
/*  12-31-2009     Yuxin Zhou               Modified comments and         */
/*                                          added multihome support,      */
/*                                          resulting in version 5.3      */
/****************************************************************************/
void  _nx_ram_network_driver_output(NX_IP *ip_ptr, NX_PACKET *packet_ptr)
{

NX_IP       *next_ip;
NX_PACKET   *packet_copy;
ULONG       destination_address_msw;
ULONG       destination_address_lsw;
UINT        old_threshold;


#ifdef NX_DEBUG_PACKET
UCHAR       *ptr;
UINT        j, i;

    ptr =  packet_ptr -> nx_packet_prepend_ptr;
    printf("Ethernet Packet: ");
    for (j = 0; j < 6; j++)
        printf("%02X", *ptr++);
    printf(" ");
    for (j = 0; j < 6; j++)
        printf("%02X", *ptr++);
    printf(" %02X", *ptr++);
    printf("%02X ", *ptr++);

    i = 0;
    for (j = 0; j < (packet_ptr -> nx_packet_length - NX_ETHERNET_SIZE); j++)
    {
        printf("%02X", *ptr++);
        i++;
        if (i > 3)
        {
            i = 0;
            printf(" ");
        }
    }
    printf("\n");


#endif

    /* Pickup the destination IP address from the packet_ptr.  */
    destination_address_msw =  (ULONG) *(packet_ptr -> nx_packet_prepend_ptr);
    destination_address_msw =  (destination_address_msw << 8) | (ULONG) *(packet_ptr ->
                                nx_packet_prepend_ptr+1);
    destination_address_lsw =  (ULONG) *(packet_ptr -> nx_packet_prepend_ptr+2);
    destination_address_lsw =  (destination_address_lsw << 8) | (ULONG) *(packet_ptr ->
                                nx_packet_prepend_ptr+3);
    destination_address_lsw =  (destination_address_lsw << 8) | (ULONG) *(packet_ptr ->
                                nx_packet_prepend_ptr+4);
    destination_address_lsw =  (destination_address_lsw << 8) | (ULONG) *(packet_ptr ->
                                nx_packet_prepend_ptr+5);

    /* Disable preemption.  */
    tx_thread_preemption_change(tx_thread_identify(), 0, &old_threshold);

    /* Loop through all instances of created IPs to see who gets the packet.  */
    next_ip =  ip_ptr -> nx_ip_created_next;

    while (next_ip != ip_ptr)
    {

        /* Check for broadcast or a match with this IP destination.  */
        if (((destination_address_msw == ((ULONG) 0x0000FFFF)) &&
             (destination_address_lsw == ((ULONG) 0xFFFFFFFF))) ||
```

```
           ((destination_address_msw ==
             next_ip ->
           nx_ip_interface[0].nx_interface_physical_address_msw) &&
            (destination_address_lsw ==
             next_ip ->nx_ip_interface[0].nx_interface_physical_address_lsw)))
       {

           /* Make a copy of packet for the forwarding.  */
           if (nx_packet_copy(packet_ptr, &packet_copy, next_ip ->
                              nx_ip_default_packet_pool, NX_NO_WAIT))
           {

               /* Error, no point in continuing.  */
               nx_packet_transmit_release(packet_ptr);
               return;
           }

           /* For now, route the packet to the next created IP instance.  */
           _nx_ram_network_driver_receive(next_ip, packet_copy);
       }

       /* Move to the next IP instance.  */
       next_ip =  next_ip -> nx_ip_created_next;
   }

   /* Restore the packet prepend pointer.  In real hardware environments, this is typically
      done after a transmit complete interrupt.  */
   packet_ptr -> nx_packet_prepend_ptr =  packet_ptr -> nx_packet_prepend_ptr + NX_ETHERNET_SIZE;

   /* Adjust the packet length.  */
   packet_ptr -> nx_packet_length =  packet_ptr -> nx_packet_length - NX_ETHERNET_SIZE;

   /* Release the packet.  */
   nx_packet_transmit_release(packet_ptr);

   /* Restore preemption.  */
   tx_thread_preemption_change(tx_thread_identify(), old_threshold, &old_threshold);
}


/**************************************************************************/
/*                                                                        */
/*  FUNCTION                                               RELEASE        */
/*                                                                        */
/*    _nx_ram_network_driver_receive                      PORTABLE C      */
/*                                                           5.3          */
/*  AUTHOR                                                                */
/*                                                                        */
/*    William E. Lamie, Express Logic, Inc.                               */
/*                                                                        */
/*  DESCRIPTION                                                           */
/*                                                                        */
/*    This function processing incoming packets.  In the RAM network      */
/*    driver, the incoming packets are coming from the RAM driver output  */
/*    routine.  In real hardware settings, this routine would be called   */
/*    from the receive packet ISR.                                        */
/*                                                                        */
/*  INPUT                                                                 */
/*                                                                        */
/*    ip_ptr                              Pointer to IP protocol block    */
/*    packet_ptr                          Packet pointer                  */
/*                                                                        */
/*  OUTPUT                                                                */
/*                                                                        */
/*    None                                                                */
/*                                                                        */
/*  CALLS                                                                 */
/*                                                                        */
/*    _nx_ip_packet_receive               IP receive packet processing    */
/*    _nx_ip_packet_deferred_receive      IP deferred receive packet      */
/*                                          processing                    */
/*    _nx_arp_packet_deferred_receive     ARP receive processing          */
/*    _nx_rarp_packet_deferred_receive    RARP receive processing         */
```

```
/*    nx_packet_release                     Packet release                */
/*                                                                        */
/*  CALLED BY                                                             */
/*                                                                        */
/*    NetX IP processing                                                  */
/*                                                                        */
/*  RELEASE HISTORY                                                       */
/*                                                                        */
/*    DATE              NAME                      DESCRIPTION             */
/*                                                                        */
/*  12-12-2005     William E. Lamie         Initial Version 5.0           */
/*  08-09-2007     William E. Lamie         Modified comments resulting   */
/*                                          in version 5.1                */
/*  07-04-2009     William E. Lamie         Modified comments, resulting  */
/*                                          in version 5.2                */
/*  12-31-2009     Yuxin Zhou               Modified comments resulting   */
/*                                          in version 5.3                */
/**************************************************************************/
void  _nx_ram_network_driver_receive(NX_IP *ip_ptr, NX_PACKET *packet_ptr)
{

UINT    packet_type;

    /* Pickup the packet header to determine where the packet needs to be
       sent.  */
    packet_type =  (((UINT) (*(packet_ptr -> nx_packet_prepend_ptr+12))) << 8) |
                    ((UINT) (*(packet_ptr -> nx_packet_prepend_ptr+13)));

    /* Pickup the interface pointer.  */
    packet_ptr -> nx_packet_ip_interface = &(ip_ptr -> nx_ip_interface[0]);

    /* Route the incoming packet according to its ethernet type.  */
    if (packet_type == NX_ETHERNET_IP)
    {

        /* Note:  The length reported by some Ethernet hardware includes bytes after the packet
           as well as the Ethernet header.  In some cases, the actual packet length after the
           Ethernet header should be derived from the length in the IP header (lower 16 bits of
           the first 32-bit word).  */

        /* Clean off the Ethernet header.  */
        packet_ptr -> nx_packet_prepend_ptr =  packet_ptr ->
                                            nx_packet_prepend_ptr + NX_ETHERNET_SIZE;

        /* Adjust the packet length.  */
        packet_ptr -> nx_packet_length =  packet_ptr -> nx_packet_length - NX_ETHERNET_SIZE;

        /* Route to the ip receive function.  */
#ifdef NX_DEBUG_PACKET
        printf("NetX RAM Driver IP Packet Receive - %s\n", ip_ptr -> nx_ip_name);
#endif

#ifdef NX_DIRECT_ISR_CALL
        _nx_ip_packet_receive(ip_ptr, packet_ptr);
#else
        _nx_ip_packet_deferred_receive(ip_ptr, packet_ptr);
#endif
    }
    else if (packet_type == NX_ETHERNET_ARP)
    {

        /* Clean off the Ethernet header.  */
        packet_ptr -> nx_packet_prepend_ptr =  packet_ptr ->
                                              nx_packet_prepend_ptr + NX_ETHERNET_SIZE;

        /* Adjust the packet length.  */
        packet_ptr -> nx_packet_length =  packet_ptr -> nx_packet_length - NX_ETHERNET_SIZE;

        /* Route to the ARP receive function.  */
#ifdef NX_DEBUG
```

```
        printf("NetX RAM Driver ARP Receive - %s\n", ip_ptr -> nx_ip_name);
#endif
        _nx_arp_packet_deferred_receive(ip_ptr, packet_ptr);

    }
    else if (packet_type == NX_ETHERNET_RARP)
    {

        /* Clean off the Ethernet header.  */
        packet_ptr -> nx_packet_prepend_ptr =  packet_ptr -> nx_packet_prepend_ptr +
                                               NX_ETHERNET_SIZE;

        /* Adjust the packet length.  */
        packet_ptr -> nx_packet_length =  packet_ptr -> nx_packet_length - NX_ETHERNET_SIZE;

        /* Route to the RARP receive function.  */
#ifdef NX_DEBUG
        printf("NetX RAM Driver RARP Receive - %s\n", ip_ptr -> nx_ip_name);
#endif
        _nx_rarp_packet_deferred_receive(ip_ptr, packet_ptr);
    }
    else
    {

        /* Invalid ethernet header... release the packet.  */
        nx_packet_release(packet_ptr);
    }
}
```

APPENDIX **A**

NetX Services

- Address Resolution Protocol (ARP) 376
- Internet Control Message Protocol (ICMP) 376
- Internet Group Management Protocol (IGMP) 377
- Internet Protocol (IP) 377
- Packet Management 378
- Reverse Address Resolution Protocol (RARP) 379
- System Management 379
- Transmission Control Protocol (TCP) 379
- User Datagram Protocol (UDP) 381

Address Resolution Protocol (ARP)

```
UINT      nx_arp_dynamic_entries_invalidate(NX_IP *ip_ptr);

UINT      nx_arp_dynamic_entry_set(NX_IP *ip_ptr, ULONG
             ip_address, ULONG physical_msw, ULONG physical_lsw);

UINT      nx_arp_enable(NX_IP *ip_ptr, VOID *arp_cache_memory,
             ULONG arp_cache_size);

UINT      nx_arp_gratuitous_send(NX_IP *ip_ptr,
             VOID (*response_handler)(NX_IP *ip_ptr,
             NX_PACKET *packet_ptr));

UINT      nx_arp_hardware_address_find(NX_IP *ip_ptr,
             ULONG ip_address, ULONG*physical_msw,
             ULONG *physical_lsw);

UINT      nx_arp_info_get(NX_IP *ip_ptr, ULONG
             *arp_requests_sent, ULONG*arp_requests_received,
             ULONG *arp_responses_sent,
             ULONG*arp_responses_received,
             ULONG *arp_dynamic_entries,
             ULONG *arp_static_entries,
             ULONG *arp_aged_entries,
             ULONG *arp_invalid_messages);

UINT      nx_arp_ip_address_find(NX_IP *ip_ptr,
             ULONG *ip_address, ULONG physical_msw,
             ULONG physical_lsw);

UINT      nx_arp_static_entries_delete(NX_IP *ip_ptr);

UINT      nx_arp_static_entry_create(NX_IP *ip_ptr,
             ULONG ip_address,
             ULONG physical_msw, ULONG physical_lsw);

UINT      nx_arp_static_entry_delete(NX_IP *ip_ptr,
             ULONG ip_address, ULONG physical_msw,
             ULONG physical_lsw);
```

Internet Control Message Protocol (ICMP)

```
UINT      nx_icmp_enable(NX_IP *ip_ptr);

UINT      nx_icmp_info_get(NX_IP *ip_ptr, ULONG *pings_sent,
             ULONG *ping_timeouts, ULONG *ping_threads_suspended,
             ULONG *ping_responses_received,
             ULONG *icmp_checksum_errors,
             ULONG *icmp_unhandled_messages);

UINT      nx_icmp_ping(NX_IP *ip_ptr,
             ULONG ip_address, CHAR *data,
             ULONG data_size, NX_PACKET **response_ptr,
             ULONG wait_option);
```

Internet Group Management Protocol (IGMP)

```
UINT      nx_igmp_enable(NX_IP *ip_ptr);

UINT      nx_igmp_info_get(NX_IP *ip_ptr, ULONG
             *igmp_reports_sent, ULONG *igmp_queries_received,
             ULONG *igmp_checksum_errors,
             ULONG *current_groups_joined);

UINT      nx_igmp_loopback_disable(NX_IP *ip_ptr);

UINT      nx_igmp_loopback_enable(NX_IP *ip_ptr);

UINT      nx_igmp_multicast_interface_join(NX_IP *ip_ptr,
             ULONG group_address, UINT interface_index);

UINT      nx_igmp_multicast_join(NX_IP *ip_ptr,
             ULONG group_address);

UINT      nx_igmp_multicast_leave(NX_IP *ip_ptr,
             ULONG group_address);
```

Internet Protocol (IP)

```
UINT      nx_ip_address_change_notify(NX_IP *ip_ptr,
             VOID (*change_notify)(NX_IP *, VOID *),
             VOID *additional_info);

UINT      nx_ip_address_get(NX_IP *ip_ptr, ULONG *ip_address,
             ULONG *network_mask);

UINT      nx_ip_address_set(NX_IP *ip_ptr, ULONG ip_address,
             ULONG network_mask);

UINT      nx_ip_create(NX_IP *ip_ptr, CHAR *name,
             ULONG ip_address,
             ULONG network_mask, NX_PACKET_POOL *default_pool,
             VOID (*ip_network_driver)(NX_IP_DRIVER *),
             VOID *memory_ptr, ULONG memory_size, UINT priority);

UINT      nx_ip_delete(NX_IP *ip_ptr);

UINT      nx_ip_driver_direct_command(NX_IP *ip_ptr, UINT
             command, ULONG *return_value_ptr);

UINT      nx_ip_forwarding_disable(NX_IP *ip_ptr);

UINT      nx_ip_forwarding_enable(NX_IP *ip_ptr);

UINT      nx_ip_fragment_disable(NX_IP *ip_ptr);

UINT      nx_ip_fragment_enable(NX_IP *ip_ptr);

UINT      nx_ip_gateway_address_set(NX_IP *ip_ptr,
             ULONG ip_address);

UINT      nx_ip_info_get(NX_IP *ip_ptr,
             ULONG *ip_total_packets_sent,
             ULONG *ip_total_bytes_sent,
             ULONG *ip_total_packets_received,
             ULONG *ip_total_bytes_received,
             ULONG *ip_invalid_packets,
```

```
                                ULONG *ip_receive_packets_dropped,
                                ULONG *ip_receive_checksum_errors,
                                ULONG *ip_send_packets_dropped,
                                ULONG *ip_total_fragments_sent,
                                ULONG *ip_total_fragments_received);

UINT      nx_ip_interface_address_get(NX_IP *ip_ptr, ULONG
                                interface_index, ULONG *ip_address, ULONG *network_mask);

UINT      nx_ip_interface_address_set(NX_IP *ip_ptr,
                                ULONG interface_index, ULONG ip_address, ULONG network_mask);

UINT      nx_ip_interface_attach(NX_IP *ip_ptr, CHAR* interface_name,
                                ULONG ip_address, ULONG network_mask,
                                VOID (*ip_link_driver)(struct NX_IP_DRIVER_STRUCT *));

UINT      nx_ip_interface_info_get(NX_IP *ip_ptr, UINT interface_index,
                                CHAR **interface_name, ULONG *ip_address,
                                ULONG *network_mask, ULONG *mtu_size,
                                ULONG *phsyical_address_msw, ULONG *physical_address_lsw);

UINT      nx_ip_interface_status_check(NX_IP *ip_ptr,
                                UINT interface_index, ULONG needed_status,
                                ULONG *actual_status, ULONG wait_option);

UINT      nx_ip_raw_packet_disable(NX_IP *ip_ptr);

UINT      nx_ip_raw_packet_enable(NX_IP *ip_ptr);

UINT      nx_ip_raw_packet_interface_send(NX_IP *ip_ptr,
                                NX_PACKET *packet_ptr, ULONG destination_ip,
                                UINT interface_index, ULONG type_of_service);

UINT      nx_ip_raw_packet_receive(NX_IP *ip_ptr,
                                NX_PACKET **packet_ptr,
                                ULONG wait_option);

UINT      nx_ip_raw_packet_send(NX_IP *ip_ptr,
                                NX_PACKET *packet_ptr,
                                ULONG destination_ip, ULONG type_of_service);

UINT      nx_ip_static_route_add(NX_IP *ip_ptr, ULONG network_address,
                                ULONG net_mask, ULONG next_hop);

UINT      nx_ip_static_route_delete(NX_IP *ip_ptr, ULONG network_address,
                                ULONG net_mask);

UINT      nx_ip_status_check(NX_IP *ip_ptr, ULONG needed_status, ULONG
                                *actual_status, ULONG wait_option);
```

Packet Management

```
UINT      nx_packet_allocate(NX_PACKET_POOL *pool_ptr,
                                NX_PACKET **packet_ptr, ULONG packet_type,
                                ULONG wait_option);

UINT      nx_packet_copy(NX_PACKET *packet_ptr,
                                NX_PACKET **new_packet_ptr, NX_PACKET_POOL *pool_ptr,
                                ULONG wait_option);
```

```
UINT      nx_packet_data_append(NX_PACKET *packet_ptr,
             VOID *data_start, ULONG data_size,
             NX_PACKET_POOL *pool_ptr, ULONG wait_option);

UINT      nx_packet_data_extract_offset(NX_PACKET *packet_ptr,
             ULONG offset, VOID *buffer_start, ULONG buffer_length,
             ULONG *bytes_copied);

UINT      nx_packet_data_retrieve(NX_PACKET *packet_ptr,
             VOID *buffer_start, ULONG *bytes_copied);

UINT      nx_packet_length_get(NX_PACKET *packet_ptr, ULONG *length);

UINT      nx_packet_pool_create(NX_PACKET_POOL *pool_ptr,
             CHAR *name, ULONG block_size, VOID *memory_ptr, ULONG
             memory_size);

UINT      nx_packet_pool_delete(NX_PACKET_POOL *pool_ptr);

UINT      nx_packet_pool_info_get(NX_PACKET_POOL *pool_ptr, ULONG
             *total_packets, ULONG *free_packets,
             ULONG *empty_pool_requests,
             ULONG *empty_pool_suspensions,
             ULONG *invalid_packet_releases);

UINT      nx_packet_release(NX_PACKET *packet_ptr);

UINT      nx_packet_transmit_release(NX_PACKET *packet_ptr);
```

Reverse Address Resolution Protocol (RARP)

```
UINT      nx_rarp_disable(NX_IP *ip_ptr);

UINT      nx_rarp_enable(NX_IP *ip_ptr);

UINT      nx_rarp_info_get(NX_IP *ip_ptr,
             ULONG *rarp_requests_sent,
             ULONG *rarp_responses_received,
             ULONG *rarp_invalid_messages);
```

System Management

```
VOID      nx_system_initialize(VOID);
```

Transmission Control Protocol (TCP)

```
UINT      nx_tcp_client_socket_bind(NX_TCP_SOCKET *socket_ptr, UINT
             port, ULONG wait_option);

UINT      nx_tcp_client_socket_connect(NX_TCP_SOCKET *socket_ptr, ULONG
             server_ip, UINT server_port,
             ULONG wait_option);

UINT      nx_tcp_client_socket_port_get(NX_TCP_SOCKET *socket_ptr, UINT
             *port_ptr);

UINT      nx_tcp_client_socket_unbind(NX_TCP_SOCKET *socket_ptr);
```

```
UINT      nx_tcp_enable(NX_IP *ip_ptr);

UINT      nx_tcp_free_port_find(NX_IP *ip_ptr, UINT port,
             UINT *free_port_ptr);

UINT      nx_tcp_info_get(NX_IP *ip_ptr, ULONG *tcp_packets_sent,
             ULONG *tcp_bytes_sent, ULONG *tcp_packets_received,
             ULONG *tcp_bytes_received, ULONG
             *tcp_invalid_packets, ULONG
             *tcp_receive_packets_dropped,
             ULONG *tcp_checksum_errors,ULONG *tcp_connections,
             ULONG *tcp_disconnections,
             ULONG *tcp_connections_dropped,
             ULONG*tcp_retransmit_packets);

UINT      nx_tcp_server_socket_accept(NX_TCP_SOCKET *socket_ptr,
             ULONG wait_option);

UINT      nx_tcp_server_socket_listen(NX_IP *ip_ptr,
             UINT port, NX_TCP_SOCKET *socket_ptr,
             UINT listen_queue_size,
             VOID (*tcp_listen_callback)(NX_TCP_SOCKET
             *socket_ptr, UINT port));

UINT      nx_tcp_server_socket_relisten(NX_IP *ip_ptr,
             UINT port, NX_TCP_SOCKET *socket_ptr);

UINT      nx_tcp_server_socket_unaccept(NX_TCP_SOCKET
             *socket_ptr);

UINT      nx_tcp_server_socket_unlisten(NX_IP *ip_ptr, UINT
             port);

UINT      nx_tcp_socket_bytes_available(NX_TCP_SOCKET
             *socket_ptr, ULONG *bytes_available);

UINT      nx_tcp_socket_create(NX_IP *ip_ptr,
             NX_TCP_SOCKET *socket_ptr, CHAR *name,
             ULONG type_of_service, ULONG fragment,
             UINT time_to_live, ULONG window_size,
             VOID (*tcp_urgent_data_callback)(NX_TCP_SOCKET
             *socket_ptr),
             VOID (*tcp_disconnect_callback)(NX_TCP_SOCKET
             *socket_ptr));

UINT      nx_tcp_socket_delete(NX_TCP_SOCKET *socket_ptr);

UINT      nx_tcp_socket_disconnect(NX_TCP_SOCKET *socket_ptr,
             ULONG wait_option);

UINT      nx_tcp_socket_info_get(NX_TCP_SOCKET *socket_ptr,
             ULONG *tcp_packets_sent, ULONG *tcp_bytes_sent,
             ULONG *tcp_packets_received, ULONG
             *tcp_bytes_received,
             ULONG *tcp_retransmit_packets, ULONG
             *tcp_packets_queued,
             ULONG *tcp_checksum_errors, ULONG *tcp_socket_state,
             ULONG *tcp_transmit_queue_depth, ULONG
             *tcp_transmit_window,
             ULONG *tcp_receive_window);
```

```
UINT      nx_tcp_socket_mss_get(NX_TCP_SOCKET *socket_ptr,
             ULONG *mss);

UINT      nx_tcp_socket_mss_peer_get(NX_TCP_SOCKET *socket_ptr,
             ULONG *peer_mss);

UINT      nx_tcp_socket_mss_set(NX_TCP_SOCKET *socket_ptr,
             ULONG mss);

UINT      nx_tcp_socket_peer_info_get(NX_TCP_SOCKET *socket_ptr,
             ULONG *peer_ip_address, ULONG *peer_port);

UINT      nx_tcp_socket_receive(NX_TCP_SOCKET *socket_ptr,
             NX_PACKET **packet_ptr, ULONG wait_option);

UINT      nx_tcp_socket_receive_notify(NX_TCP_SOCKET
             *socket_ptr, VOID
             (*tcp_receive_notify)(NX_TCP_SOCKET *socket_ptr));

UINT      nx_tcp_socket_send(NX_TCP_SOCKET *socket_ptr,
             NX_PACKET *packet_ptr, ULONG wait_option);

UINT      nx_tcp_socket_state_wait(NX_TCP_SOCKET *socket_ptr,
             UINT desired_state, ULONG wait_option);

UINT      nx_tcp_socket_transmit_configure(NX_TCP_SOCKET
             *socket_ptr, ULONG max_queue_depth, ULONG timeout,
             ULONG max_retries, ULONG timeout_shift);

UINT      nx_tcp_socket_window_update_notify_set
             (NX_TCP_SOCKET *socket_ptr,
             VOID (*tcp_window_update_notify)
             (NX_TCP_SOCKET *socket_ptr));
```

User Datagram Protocol (UDP)

```
UINT      nx_udp_enable(NX_IP *ip_ptr);

UINT      nx_udp_free_port_find(NX_IP *ip_ptr, UINT port,
             UINT *free_port_ptr);

UINT      nx_udp_info_get(NX_IP *ip_ptr, ULONG *udp_packets_sent,
             ULONG *udp_bytes_sent, ULONG *udp_packets_received,
             ULONG *udp_bytes_received,
             ULONG *udp_invalid_packets,
             ULONG *udp_receive_packets_dropped,
             ULONG *udp_checksum_errors);

UINT      nx_udp_packet_info_extract(NX_PACKET *packet_ptr,
             ULONG *ip_address, UINT *protocol, UINT *port,
             UINT *interface_index);

UINT      nx_udp_socket_bind(NX_UDP_SOCKET *socket_ptr,
             UINT  port, ULONG wait_option);

UINT      nx_udp_socket_bytes_available(NX_UDP_SOCKET
             *socket_ptr, ULONG *bytes_available);

UINT      nx_udp_socket_checksum_disable(NX_UDP_SOCKET
             *socket_ptr);
```

```
UINT      nx_udp_socket_checksum_enable(NX_UDP_SOCKET
             *socket_ptr);

UINT      nx_udp_socket_create(NX_IP *ip_ptr, NX_UDP_SOCKET
             *socket_ptr, CHAR *name, ULONG type_of_service,
             ULONG fragment,
             UINT time_to_live, ULONG queue_maximum);

UINT      nx_udp_socket_delete(NX_UDP_SOCKET *socket_ptr);

UINT      nx_udp_socket_info_get(NX_UDP_SOCKET *socket_ptr,
             ULONG *udp_packets_sent, ULONG *udp_bytes_sent,
             ULONG *udp_packets_received, ULONG
             *udp_bytes_received,
             ULONG *udp_packets_queued,
             ULONG *udp_receive_packets_dropped,
             ULONG *udp_checksum_errors);

UINT      nx_udp_socket_interface_send(NX_UDP_SOCKET
             *socket_ptr, NX_PACKET *packet_ptr, ULONG
             ip_address, UINT port, UINT interface_index);

UINT      nx_udp_socket_port_get(NX_UDP_SOCKET *socket_ptr,
             UINT *port_ptr);

UINT      nx_udp_socket_receive(NX_UDP_SOCKET *socket_ptr,
             NX_PACKET **packet_ptr, ULONG wait_option);

UINT      nx_udp_socket_receive_notify(NX_UDP_SOCKET
             *socket_ptr, VOID
             (*udp_receive_notify)(NX_UDP_SOCKET *socket_ptr));

UINT      nx_udp_socket_send(NX_UDP_SOCKET *socket_ptr,
             NX_PACKET *packet_ptr, ULONG ip_address, UINT port);

UINT      nx_udp_socket_unbind(NX_UDP_SOCKET *socket_ptr);

UINT      nx_udp_source_extract(NX_PACKET *packet_ptr,
             ULONG *ip_address, UINT *port);
```

APPENDIX **B**

NetX Constants

- Alphabetic Listing 384
- Listings by Value 393

Alphabetic Listing

NX_ALL_HOSTS_ADDRESS	0xFE000001
NX_ALREADY_BOUND	0x22
NX_ALREADY_ENABLED	0x15
NX_ALREADY_RELEASED	0x31
NX_ALREADY_SUSPENDED	0x40
NX_ANY_PORT	0
NX_ARP_DEBUG_LOG_SIZE	100
NX_ARP_EXPIRATION_RATE	0
NX_ARP_HARDWARE_SIZE	0x06
NX_ARP_HARDWARE_TYPE	0x0001
NX_ARP_MAX_QUEUE_DEPTH	4
NX_ARP_MAXIMUM_RETRIES	18
NX_ARP_MESSAGE_SIZE	28
NX_ARP_OPTION_REQUEST	0x0001
NX_ARP_OPTION_RESPONSE	0x0002
NX_ARP_PROTOCOL_SIZE	0x04
NX_ARP_PROTOCOL_TYPE	0x0800
NX_ARP_TIMER_ERROR	0x18
NX_ARP_UPDATE_RATE	10
NX_CALLER_ERROR	0x11
NX_CARRY_BIT	0x10000
NX_CONNECTION_PENDING	0x48
NX_DELETE_ERROR	0x10
NX_DELETED	0x05
NX_DISCONNECT_FAILED	0x41
NX_DONT_FRAGMENT	0x00004000
NX_DRIVER_TX_DONE	0xDDDDDDDD
NX_DUPLICATE_LISTEN	0x34
NX_ENTRY_NOT_FOUND	0x16
NX_FALSE	0
NX_FOREVER	1

NX_FRAG_OFFSET_MASK	0x00001FFF
NX_FRAGMENT_OKAY	0x00000000
NX_ICMP_ADDRESS_MASK_REP_TYPE	18
NX_ICMP_ADDRESS_MASK_REQ_TYPE	17
NX_ICMP_DEBUG_LOG_SIZE	100
NX_ICMP_DEST_UNREACHABLE_TYPE	3
NX_ICMP_ECHO_REPLY_TYPE	0
NX_ICMP_ECHO_REQUEST_TYPE	8
NX_ICMP_FRAMENT_NEEDED_CODE	4
NX_ICMP_HOST_PROHIBIT_CODE	10
NX_ICMP_HOST_SERVICE_CODE	12
NX_ICMP_HOST_UNKNOWN_CODE	7
NX_ICMP_HOST_UNREACH_CODE	1
NX_ICMP_NETWORK_PROHIBIT_CODE	9
NX_ICMP_NETWORK_SERVICE_CODE	11
NX_ICMP_NETWORK_UNKNOWN_CODE	6
NX_ICMP_NETWORK_UNREACH_CODE	0
NX_ICMP_PACKET	36
NX_ICMP_PARAMETER_PROB_TYPE	12
NX_ICMP_PORT_UNREACH_CODE	3
NX_ICMP_PROTOCOL_UNREACH_CODE	2
NX_ICMP_REDIRECT_TYPE	5
NX_ICMP_SOURCE_ISOLATED_CODE	8
NX_ICMP_SOURCE_QUENCH_TYPE	4
NX_ICMP_SOURCE_ROUTE_CODE	5
NX_ICMP_TIME_EXCEEDED_TYPE	11
NX_ICMP_TIMESTAMP_REP_TYPE	14
NX_ICMP_TIMESTAMP_REQ_TYPE	13
NX_IGMP_DEBUG_LOG_SIZE	100
NX_IGMP_HOST_RESPONSE_TYPE	0x02000000
NX_IGMP_MAX_UPDATE_TIME	10

NX_IGMP_PACKET	36
NX_IGMP_ROUTER_QUERY_TYPE	0x01000000
NX_IGMP_TYPE_MASK	0x0F000000
NX_IGMP_VERSION	0x10000000
NX_IGMP_VERSION_MASK	0xF0000000
NX_IN_PROGRESS	0x37
NX_INIT_PACKET_ID	1
NX_NOT_IMPLEMENTED	0x4A
NX_NOT_SUPPORTED	0x4B
NX_INVALID_INTERFACE	0x4C
NX_INVALID_PACKET	0x12
NX_INVALID_PORT	0x46
NX_INVALID_RELISTEN	0x47
NX_INVALID_SOCKET	0x13
NX_IP_ADDRESS_ERROR	0x21
NX_IP_ADDRESS_RESOLVED	0x0002
NX_IP_ALIGN_FRAGS	8
NX_IP_ALL_EVENTS	0xFFFFFFFF
NX_IP_ARP_ENABLED	0x0008
NX_IP_ARP_REC_EVENT	0x00000010
NX_IP_CLASS_A_HOSTID	0x00FFFFFF
NX_IP_CLASS_A_MASK	0x80000000
NX_IP_CLASS_A_NETID	0x7F000000
NX_IP_CLASS_A_TYPE	0x00000000
NX_IP_CLASS_B_HOSTID	0x0000FFFF
NX_IP_CLASS_B_MASK	0xC0000000
NX_IP_CLASS_B_NETID	0x3FFF0000
NX_IP_CLASS_B_TYPE	0x80000000
NX_IP_CLASS_C_HOSTID	0x000000FF
NX_IP_CLASS_C_MASK	0xE0000000
NX_IP_CLASS_C_NETID	0x1FFFFF00

NX_IP_CLASS_C_TYPE	0xC0000000
NX_IP_CLASS_D_GROUP	0x0FFFFFFF
NX_IP_CLASS_D_HOSTID	0x00000000
NX_IP_CLASS_D_MASK	0xF0000000
NX_IP_CLASS_D_TYPE	0xE0000000
NX_IP_DEBUG_LOG_SIZE	100
NX_IP_DONT_FRAGMENT	0x00004000
NX_IP_DRIVER_DEFERRED_EVENT	0x00000800
NX_IP_DRIVER_PACKET_EVENT	0x00000200
NX_IP_FRAGMENT_MASK	0x00003FFF
NX_IP_ICMP	0x00010000
NX_IP_ICMP_EVENT	0x00000004
NX_IP_ID	0x49502020
NX_IP_IGMP	0x00020000
NX_IP_IGMP_ENABLE_EVENT	0x00000400
NX_IP_IGMP_ENABLED	0x0040
NX_IP_IGMP_EVENT	0x00000040
NX_IP_INITIALIZE_DONE	0x0001
NX_IP_INTERNAL_ERROR	0x20
NX_IP_LENGTH_MASK	0x0F000000
NX_IP_LIMITIED_BROADCAST	0xFFFFFFFF
NX_IP_LINK_ENABLED	0x0004
NX_IP_LOOPBACK_FIRST	0x7F000000
NX_IP_LOOPBACK_LAST	0x7FFFFFFF
NX_IP_MAX_DATA	0x00080000
NX_IP_MAX_RELIABLE	0x00040000
NX_IP_MIN_COST	0x00020000
NX_IP_MIN_DELAY	0x00100000
NX_IP_MORE_FRAGMENT	0x00002000
NX_IP_MULTICAST_LOWER	0x5E000000
NX_IP_MULTICAST_MASK	0x007FFFFF

NX_IP_MULTICAST_UPPER	0x00000100
NX_IP_NORMAL	0x00000000
NX_IP_NORMAL_LENGTH	5
NX_IP_OFFSET_MASK	0x00001FFF
NX_IP_PACKET	36
NX_IP_PACKET_SIZE_MASK	0x0000FFFF
NX_IP_PERIODIC_EVENT	0x00000001
NX_IP_PERIODIC_RATE	100
NX_IP_PROTOCOL_MASK	0x00FF0000
NX_IP_RARP_COMPLETE	0x0080
NX_IP_RARP_REC_EVENT	0x00000020
NX_IP_RECEIVE_EVENT	0x00000008
NX_IP_TCP	0x00060000
NX_IP_TCP_CLEANUP_DEFERRED	0x00001000
NX_IP_TCP_ENABLED	0x0020
NX_IP_TCP_EVENT	0x00000080
NX_IP_TCP_FAST_EVENT	0x00000100
NX_IP_TIME_TO_LIVE	0x00000080
NX_IP_TIME_TO_LIVE_MASK	0xFF000000
NX_IP_TIME_TO_LIVE_SHIFT	24
NX_IP_TOS_MASK	0x00FF0000
NX_IP_UDP	0x00110000
NX_IP_UDP_ENABLED	0x0010
NX_IP_UNFRAG_EVENT	0x00000002
NX_IP_VERSION	0x45000000
NX_LINK_ARP_RESPONSE_SEND	6
NX_LINK_ARP_SEND	5
NX_LINK_DEFERRED_PROCESSING	18
NX_LINK_DISABLE	3
NX_LINK_ENABLE	2
NX_LINK_GET_ALLOC_ERRORS	16

NX_LINK_GET_DUPLEX_TYPE	12
NX_LINK_GET_ERROR_COUNT	13
NX_LINK_GET_RX_COUNT	14
NX_LINK_GET_SPEED	11
NX_LINK_GET_STATUS	10
NX_LINK_GET_TX_COUNT	15
NX_LINK_INITIALIZE	1
NX_LINK_INTERFACE_ATTACH	19
NX_LINK_MULTICAST_JOIN	8
NX_LINK_MULTICAST_LEAVE	9
NX_LINK_PACKET_BROADCAST	4
NX_LINK_PACKET_SEND	0
NX_LINK_RARP_SEND	7
NX_LINK_UNINITIALIZE	17
NX_LINK_USER_COMMAND	50
NX_LOWER_16_MASK	0x0000FFFF
NX_MAX_LISTEN	0x33
NX_MAX_LISTEN_REQUESTS	10
NX_MAX_MULTICAST_GROUPS	7
NX_MAX_PORT	0xFFFF
NX_MORE_FRAGMENTS	0x00002000
NX_NO_FREE_PORTS	0x45
NX_NO_MAPPING	0x04
NX_NO_MORE_ENTRIES	0x17
NX_NO_PACKET	0x01
NX_NO_RESPONSE	0x29
NX_NO_WAIT	0
NX_NOT_BOUND	0x24
NX_NOT_CLOSED	0x35
NX_NOT_CONNECTED	0x38
NX_NOT_CREATED	0x27

NX_NOT_ENABLED	0x14
NX_NOT_IMPLEMENTED	0x4A
NX_NOT_LISTEN_STATE	0x36
NX_NOT_SUCCESSFUL	0x43
NX_NULL	0
NX_OPTION_ERROR	0x0a
NX_OVERFLOW	0x03
NX_PACKET_ALLOCATED	0xAAAAAAAA
NX_PACKET_DEBUG_LOG_SIZE	100
NX_PACKET_ENQUEUED	0xEEEEEEEE
NX_PACKET_FREE	0xFFFFFFFF
NX_PACKET_POOL_ID	0x5041434B
NX_PACKET_READY	0xBBBBBBBB
NX_PHYSICAL_HEADER	16
NX_PHYSICAL_TRAILER	4
NX_POOL_DELETED	0x30
NX_POOL_ERROR	0x06
NX_PORT_UNAVAILABLE	0x23
NX_PTR_ERROR	0x07
NX_RARP_DEBUG_LOG_SIZE	100
NX_RARP_HARDWARE_SIZE	0x06
NX_RARP_HARDWARE_TYPE	0x0001
NX_RARP_MESSAGE_SIZE	28
NX_RARP_OPTION_REQUEST	0x0003
NX_RARP_OPTION_RESPONSE	0x0004
NX_RARP_PROTOCOL_SIZE	0x04
NX_RARP_PROTOCOL_TYPE	0x0800
NX_RECEIVE_PACKET	0
NX_RESERVED_CODE0	0x19
NX_RESERVED_CODE1	0x25
NX_RESERVED_CODE2	0x32

NX_ROUTE_TABLE_MASK	0x1F
NX_ROUTE_TABLE_SIZE	32
NX_SEARCH_PORT_START	30000
NX_SHIFT_BY_16	16
NX_SIZE_ERROR	0x09
NX_SOCKET_UNBOUND	0x26
NX_SOCKETS_BOUND	0x28
NX_STILL_BOUND	0x42
NX_SUCCESS	0x00
NX_TCP_ACK_BIT	0x00100000
NX_TCP_ACK_TIMER_RATE	5
NX_TCP_CLIENT	1
NX_TCP_CLOSE_WAIT	6
NX_TCP_CLOSED	1
NX_TCP_CLOSING	9
NX_TCP_CONTROL_MASK	0x00170000
NX_TCP_DEBUG_LOG_SIZE	100
NX_TCP_EOL_KIND	0x00
NX_TCP_ESTABLISHED	5
NX_TCP_FAST_TIMER_RATE	10
NX_TCP_FIN_BIT	0x00010000
NX_TCP_FIN_WAIT_1	7
NX_TCP_FIN_WAIT_2	8
NX_TCP_HEADER_MASK	0xF0000000
NX_TCP_HEADER_SHIFT	28
NX_TCP_HEADER_SIZE	0x50000000
NX_TCP_ID	0x54435020
NX_TCP_KEEPALIVE_INITIAL	7200
NX_TCP_KEEPALIVE_RETRIES	10
NX_TCP_KEEPALIVE_RETRY	75
NX_TCP_LAST_ACK	11

NX_TCP_LISTEN_STATE	2
NX_TCP_MAXIMUM_RETRIES	10
NX_TCP_MAXIMUM_TX_QUEUE	20
NX_TCP_MSS_KIND	0x02
NX_TCP_MSS_OPTION	0x02040000
NX_TCP_MSS_SIZE	16384
NX_TCP_NOP_KIND	0x01
NX_TCP_OPTION_END	0x01010402
NX_TCP_PACKET	56
NX_TCP_PORT_TABLE_MASK	0x1F
NX_TCP_PORT_TABLE_SIZE	32
NX_TCP_PSH_BIT	0x00080000
NX_TCP_RETRY_SHIFT	0
NX_TCP_RST_BIT	0x00040000
NX_TCP_SERVER	2
NX_TCP_SYN_BIT	0x00020000
NX_TCP_SYN_HEADER	0x70000000
NX_TCP_SYN_RECEIVED	4
NX_TCP_SYN_SENT	3
NX_TCP_TIMED_WAIT	10
NX_TCP_TRANSMIT_TIMER_RATE	1
NX_TCP_URG_BIT	0x00200000
NX_TRUE	1
NX_TX_QUEUE_DEPTH	0x49
NX_UDP_DEBUG_LOG_SIZE	100
NX_UDP_ID	0x55445020
NX_UDP_PACKET	44
NX_UDP_PORT_TABLE_MASK	0x1F
NX_UDP_PORT_TABLE_SIZE	32
NX_UNDERFLOW	0x02
NX_UNHANDLED_COMMAND	0x44

NX_WAIT_ABORTED	0x1A
NX_WAIT_ERROR	0x08
NX_WAIT_FOREVER	0xFFFFFFFF
NX_WINDOW_OVERFLOW	0x39

Listings by Value

NX_ANY_PORT	0
NX_ARP_EXPIRATION_RATE	0
NX_FALSE	0
NX_ICMP_ECHO_REPLY_TYPE	0
NX_ICMP_NETWORK_UNREACH_CODE	0
NX_LINK_PACKET_SEND	0
NX_NO_WAIT	0
NX_NULL	0
NX_RECEIVE_PACKET	0
NX_TCP_RETRY_SHIFT	0
NX_SUCCESS	0x00
NX_TCP_EOL_KIND	0x00
NX_FRAGMENT_OKAY	0x00000000
NX_IP_CLASS_A_TYPE	0x00000000
NX_IP_CLASS_D_HOSTID	0x00000000
NX_IP_NORMAL	0x00000000
NX_FOREVER	1
NX_ICMP_HOST_UNREACH_CODE	1
NX_INIT_PACKET_ID	1
NX_LINK_INITIALIZE	1
NX_TCP_CLIENT	1
NX_TCP_CLOSED	1
NX_TCP_TRANSMIT_TIMER_RATE	1
NX_TRUE	1
NX_IP_PERIODIC_EVENT	0x00000001

NX_ARP_HARDWARE_TYPE	0x0001
NX_ARP_OPTION_REQUEST	0x0001
NX_IP_INITIALIZE_DONE	0x0001
NX_RARP_HARDWARE_TYPE	0x0001
NX_NO_PACKET	0x01
NX_TCP_NOP_KIND	0x01
NX_ICMP_PROTOCOL_UNREACH_CODE	2
NX_LINK_ENABLE	2
NX_TCP_LISTEN_STATE	2
NX_TCP_SERVER	2
NX_IP_UNFRAG_EVENT	0x00000002
NX_ARP_OPTION_RESPONSE	0x0002
NX_IP_ADDRESS_RESOLVED	0x0002
NX_TCP_MSS_KIND	0x02
NX_UNDERFLOW	0x02
NX_ICMP_DEST_UNREACHABLE_TYPE	3
NX_ICMP_PORT_UNREACH_CODE	3
NX_LINK_DISABLE	3
NX_TCP_SYN_SENT	3
NX_RARP_OPTION_REQUEST	0x0003
NX_OVERFLOW	0x03
NX_ARP_MAX_QUEUE_DEPTH	4
NX_ICMP_FRAMENT_NEEDED_CODE	4
NX_ICMP_SOURCE_QUENCH_TYPE	4
NX_LINK_PACKET_BROADCAST	4
NX_PHYSICAL_TRAILER	4
NX_TCP_SYN_RECEIVED	4
NX_IP_ICMP_EVENT	0x00000004
NX_IP_LINK_ENABLED	0x0004
NX_RARP_OPTION_RESPONSE	0x0004
NX_ARP_PROTOCOL_SIZE	0x04

NX_NO_MAPPING	0x04
NX_RARP_PROTOCOL_SIZE	0x04
NX_NOT_IMPLEMENTED	0x4A
NX_NOT_SUPPORTED	0x4B
NX_INVALID_INTERFACE	0x4C
NX_ICMP_REDIRECT_TYPE	5
NX_ICMP_SOURCE_ROUTE_CODE	5
NX_IP_NORMAL_LENGTH	5
NX_LINK_ARP_SEND	5
NX_TCP_ACK_TIMER_RATE	5
NX_TCP_ESTABLISHED	5
NX_DELETED	0x05
NX_ICMP_NETWORK_UNKNOWN_CODE	6
NX_LINK_ARP_RESPONSE_SEND	6
NX_TCP_CLOSE_WAIT	6
NX_ARP_HARDWARE_SIZE	0x06
NX_POOL_ERROR	0x06
NX_RARP_HARDWARE_SIZE	0x06
NX_ICMP_HOST_UNKNOWN_CODE	7
NX_LINK_RARP_SEND	7
NX_MAX_MULTICAST_GROUPS	7
NX_TCP_FIN_WAIT_1	7
NX_PTR_ERROR	0x07
NX_ICMP_ECHO_REQUEST_TYPE	8
NX_ICMP_SOURCE_ISOLATED_CODE	8
NX_IP_ALIGN_FRAGS	8
NX_LINK_MULTICAST_JOIN	8
NX_TCP_FIN_WAIT_2	8
NX_IP_RECEIVE_EVENT	0x00000008
NX_IP_ARP_ENABLED	0x0008
NX_WAIT_ERROR	0x08

NX_ICMP_NETWORK_PROHIBIT_CODE	9
NX_LINK_MULTICAST_LEAVE	9
NX_TCP_CLOSING	9
NX_SIZE_ERROR	0x09
NX_ARP_UPDATE_RATE	10
NX_ICMP_HOST_PROHIBIT_CODE	10
NX_IGMP_MAX_UPDATE_TIME	10
NX_LINK_GET_STATUS	10
NX_MAX_LISTEN_REQUESTS	10
NX_TCP_FAST_TIMER_RATE	10
NX_TCP_KEEPALIVE_RETRIES	10
NX_TCP_MAXIMUM_RETRIES	10
NX_TCP_TIMED_WAIT	10
NX_OPTION_ERROR	0x0a
NX_ICMP_NETWORK_SERVICE_CODE	11
NX_ICMP_TIME_EXCEEDED_TYPE	11
NX_LINK_GET_SPEED	11
NX_TCP_LAST_ACK	11
NX_ICMP_HOST_SERVICE_CODE	12
NX_ICMP_PARAMETER_PROB_TYPE	12
NX_LINK_GET_DUPLEX_TYPE	12
NX_ICMP_TIMESTAMP_REQ_TYPE	13
NX_LINK_GET_ERROR_COUNT	13
NX_ICMP_TIMESTAMP_REP_TYPE	14
NX_LINK_GET_RX_COUNT	14
NX_LINK_GET_TX_COUNT	15
NX_LINK_GET_ALLOC_ERRORS	16
NX_PHYSICAL_HEADER	16
NX_SHIFT_BY_16	16
NX_IP_ARP_REC_EVENT	0x00000010
NX_IP_UDP_ENABLED	0x0010

NX_DELETE_ERROR	0x10
NX_ICMP_ADDRESS_MASK_REQ_TYPE	17
NX_LINK_UNINITIALIZE	17
NX_CALLER_ERROR	0x11
NX_ARP_MAXIMUM_RETRIES	18
NX_ICMP_ADDRESS_MASK_REP_TYPE	18
NX_LINK_DEFERRED_PROCESSING	18
NX_INVALID_PACKET	0x12
NX_INVALID_SOCKET	0x13
NX_LINK_INTERFACE_ATTACH	19
NX_TCP_MAXIMUM_TX_QUEUE	20
NX_NOT_ENABLED	0x14
NX_ALREADY_ENABLED	0x15
NX_ENTRY_NOT_FOUND	0x16
NX_NO_MORE_ENTRIES	0x17
NX_IP_TIME_TO_LIVE_SHIFT	24
NX_ARP_TIMER_ERROR	0x18
NX_RESERVED_CODE0	0x19
NX_WAIT_ABORTED	0x1A
NX_ARP_MESSAGE_SIZE	28
NX_RARP_MESSAGE_SIZE	28
NX_TCP_HEADER_SHIFT	28
NX_ROUTE_TABLE_MASK	0x1F
NX_TCP_PORT_TABLE_MASK	0x1F
NX_UDP_PORT_TABLE_MASK	0x1F
NX_ROUTE_TABLE_SIZE	32
NX_TCP_PORT_TABLE_SIZE	32
NX_UDP_PORT_TABLE_SIZE	32
NX_IP_RARP_REC_EVENT	0x00000020
NX_IP_TCP_ENABLED	0x0020
NX_IP_INTERNAL_ERROR	0x20

NX_IP_ADDRESS_ERROR	0x21
NX_ALREADY_BOUND	0x22
NX_PORT_UNAVAILABLE	0x23
NX_ICMP_PACKET	36
NX_IGMP_PACKET	36
NX_IP_PACKET	36
NX_NOT_BOUND	0x24
NX_RESERVED_CODE1	0x25
NX_SOCKET_UNBOUND	0x26
NX_NOT_CREATED	0x27
NX_SOCKETS_BOUND	0x28
NX_NO_RESPONSE	0x29
NX_UDP_PACKET	44
NX_POOL_DELETED	0x30
NX_ALREADY_RELEASED	0x31
NX_LINK_USER_COMMAND	50
NX_RESERVED_CODE2	0x32
NX_MAX_LISTEN	0x33
NX_DUPLICATE_LISTEN	0x34
NX_NOT_CLOSED	0x35
NX_NOT_LISTEN_STATE	0x36
NX_IN_PROGRESS	0x37
NX_TCP_PACKET	56
NX_NOT_CONNECTED	0x38
NX_WINDOW_OVERFLOW	0x39
NX_IP_IGMP_EVENT	0x00000040
NX_IP_IGMP_ENABLED	0x0040
NX_ALREADY_SUSPENDED	0x40
NX_DISCONNECT_FAILED	0x41
NX_STILL_BOUND	0x42
NX_NOT_SUCCESSFUL	0x43

NX_UNHANDLED_COMMAND	0x44
NX_NO_FREE_PORTS	0x45
NX_INVALID_PORT	0x46
NX_INVALID_RELISTEN	0x47
NX_CONNECTION_PENDING	0x48
NX_TX_QUEUE_DEPTH	0x49
NX_TCP_KEEPALIVE_RETRY	75
NX_ARP_DEBUG_LOG_SIZE	100
NX_ICMP_DEBUG_LOG_SIZE	100
NX_IGMP_DEBUG_LOG_SIZE	100
NX_IP_DEBUG_LOG_SIZE	100
NX_IP_PERIODIC_RATE	100
NX_PACKET_DEBUG_LOG_SIZE	100
NX_RARP_DEBUG_LOG_SIZE	100
NX_TCP_DEBUG_LOG_SIZE	100
NX_UDP_DEBUG_LOG_SIZE	100
NX_IP_TCP_EVENT	0x00000080
NX_IP_TIME_TO_LIVE	0x00000080
NX_IP_RARP_COMPLETE	0x0080
NX_NOT_IMPLEMENTED	0x80
NX_IP_CLASS_C_HOSTID	0x000000FF
NX_IP_MULTICAST_UPPER	0x00000100
NX_IP_TCP_FAST_EVENT	0x00000100
NX_IP_DRIVER_PACKET_EVENT	0x00000200
NX_IP_IGMP_ENABLE_EVENT	0x00000400
NX_IP_DRIVER_DEFERRED_EVENT	0x00000800
NX_ARP_PROTOCOL_TYPE	0x0800
NX_RARP_PROTOCOL_TYPE	0x0800
NX_IP_TCP_CLEANUP_DEFERRED	0x00001000
NX_TCP_KEEPALIVE_INITIAL	7200
NX_FRAG_OFFSET_MASK	0x00001FFF

NX_IP_OFFSET_MASK	0x00001FFF
NX_IP_MORE_FRAGMENT	0x00002000
NX_MORE_FRAGMENTS	0x00002000
NX_IP_FRAGMENT_MASK	0x00003FFF
NX_TCP_MSS_SIZE	16384
NX_DONT_FRAGMENT	0x00004000
NX_IP_DONT_FRAGMENT	0x00004000
NX_SEARCH_PORT_START	30000
NX_IP_CLASS_B_HOSTID	0x0000FFFF
NX_IP_PACKET_SIZE_MASK	0x0000FFFF
NX_LOWER_16_MASK	0x0000FFFF
NX_MAX_PORT	0xFFFF
NX_IP_ICMP	0x00010000
NX_TCP_FIN_BIT	0x00010000
NX_CARRY_BIT	0x10000
NX_IP_IGMP	0x00020000
NX_IP_MIN_COST	0x00020000
NX_TCP_SYN_BIT	0x00020000
NX_IP_MAX_RELIABLE	0x00040000
NX_TCP_RST_BIT	0x00040000
NX_IP_TCP	0x00060000
NX_IP_MAX_DATA	0x00080000
NX_TCP_PSH_BIT	0x00080000
NX_IP_MIN_DELAY	0x00100000
NX_TCP_ACK_BIT	0x00100000
NX_IP_UDP	0x00110000
NX_TCP_CONTROL_MASK	0x00170000
NX_TCP_URG_BIT	0x00200000
NX_IP_MULTICAST_MASK	0x007FFFFF
NX_IP_PROTOCOL_MASK	0x00FF0000
NX_IP_TOS_MASK	0x00FF0000

NX_IGMP_ROUTER_QUERY_TYPE	0x01000000
NX_TCP_OPTION_END	0x01010402
NX_IGMP_HOST_RESPONSE_TYPE	0x02000000
NX_TCP_MSS_OPTION	0x02040000
NX_IGMP_TYPE_MASK	0x0F000000
NX_IP_LENGTH_MASK	0x0F000000
NX_IP_CLASS_A_HOSTID	0x00FFFFFF
NX_IP_CLASS_D_GROUP	0x0FFFFFFF
NX_IGMP_VERSION	0x10000000
NX_IP_CLASS_C_NETID	0x1FFFFF00
NX_IP_CLASS_B_NETID	0x3FFF0000
NX_IP_VERSION	0x45000000
NX_IP_ID	0x49502020
NX_TCP_HEADER_SIZE	0x50000000
NX_PACKET_POOL_ID	0x5041434B
NX_TCP_ID	0x54435020
NX_UDP_ID	0x55445020
NX_IP_MULTICAST_LOWER	0x5E000000
NX_IP_CLASS_A_NETID	0x7F000000
NX_TCP_SYN_HEADER	0x70000000
NX_IP_LOOPBACK_FIRST	0x7F000000
NX_IP_LOOPBACK_LAST	0x7FFFFFFF
NX_IP_CLASS_A_MASK	0x80000000
NX_IP_CLASS_B_TYPE	0x80000000
NX_PACKET_ALLOCATED	0xAAAAAAAA
NX_PACKET_READY	0xBBBBBBBB
NX_IP_CLASS_B_MASK	0xC0000000
NX_IP_CLASS_C_TYPE	0xC0000000
NX_DRIVER_TX_DONE	0xDDDDDDDD
NX_IP_CLASS_C_MASK	0xE0000000
NX_IP_CLASS_D_TYPE	0xE0000000

NX_PACKET_ENQUEUED	0xEEEEEEEE
NX_IGMP_VERSION_MASK	0xF0000000
NX_IP_CLASS_D_MASK	0xF0000000
NX_TCP_HEADER_MASK	0xF0000000
NX_ALL_HOSTS_ADDRESS	0xFE000001
NX_IP_TIME_TO_LIVE_MASK	0xFF000000
NX_IP_ALL_EVENTS	0xFFFFFFFF
NX_IP_LIMITIED_BROADCAST	0xFFFFFFFF
NX_PACKET_FREE	0xFFFFFFFF
NX_WAIT_FOREVER	0xFFFFFFFF

APPENDIX C

NetX Data Types

- NX_ARP 404
- NX_INTERFACE 404
- NX_IP 407
- NX_IP_DRIVER 407
- NX_IP_ROUTING_ENTRY 407
- NX_PACKET 407
- NX_PACKET_POOL 407
- NX_TCP_LISTEN 408
- NX_TCP_SOCKET 409
- NX_UDP_SOCKET 409

```
typedef struct NX_ARP_STRUCT
{
    UINT                                    nx_arp_route_static;
    UINT                                    nx_arp_entry_next_update;
    UINT                                    nx_arp_retries;
    struct NX_ARP_STRUCT                    *nx_arp_pool_next,
                                            *nx_arp_pool_previous;
    struct NX_ARP_STRUCT                    *nx_arp_active_next,
                                            *nx_arp_active_previous,
                                            **nx_arp_active_list_head;
    ULONG                                   nx_arp_ip_address;
    ULONG                                   nx_arp_physical_address_msw;
    ULONG                                   nx_arp_physical_address_lsw;
    struct NX_INTERFACE_STRUCT              *nx_arp_ip_interface;
    struct NX_PACKET_STRUCT                 *nx_arp_packets_waiting;
} NX_ARP;


typedef struct NX_INTERFACE_STRUCT
{
    CHAR        *nx_interface_name;
    UCHAR       nx_interface_valid;
    UCHAR       nx_interface_mapping_needed;
    UCHAR       nx_interface_link_up;
    UCHAR       nx_interface_reserved;
    struct NX_IP_STRUCT *nx_interface_ip_instance;
    ULONG       nx_interface_physical_address_msw;
    ULONG       nx_interface_physical_address_lsw;
    ULONG       nx_interface_ip_address;
    ULONG       nx_interface_ip_network_mask;
    ULONG       nx_interface_ip_network;
    ULONG       nx_interface_ip_mtu_size;
    VOID        *nx_interface_additional_link_info;
    VOID        (*nx_interface_link_driver_entry)(struct NX_IP_DRIVER_STRUCT *);
} NX_INTERFACE;


typedef struct NX_IP_STRUCT
{
    ULONG                                   nx_ip_id;
    CHAR                                    *nx_ip_name;
    ULONG                                   nx_ip_gateway_address;
    struct NX_INTERFACE_STRUCT              *nx_ip_gateway_interface;
    ULONG                                   nx_ip_total_packet_send_requests;
    ULONG                                   nx_ip_total_packets_sent;
    ULONG                                   nx_ip_total_bytes_sent;
    ULONG                                   nx_ip_total_packets_received;
    ULONG                                   nx_ip_total_packets_delivered;
    ULONG                                   nx_ip_total_bytes_received;
    ULONG                                   nx_ip_packets_forwarded;
    ULONG                                   nx_ip_packets_reassembled;
    ULONG                                   nx_ip_reassembly_failures;
    ULONG                                   nx_ip_invalid_packets;
    ULONG                                   nx_ip_invalid_transmit_packets;
    ULONG                                   nx_ip_invalid_receive_address;
    ULONG                                   nx_ip_unknown_protocols_received;
    ULONG                                   nx_ip_transmit_resource_errors;
    ULONG                                   nx_ip_transmit_no_route_errors;
    ULONG                                   nx_ip_receive_packets_dropped;
    ULONG                                   nx_ip_receive_checksum_errors;
    ULONG                                   nx_ip_send_packets_dropped;
    ULONG                                   nx_ip_total_fragment_requests;
    ULONG                                   nx_ip_successful_fragment_requests;
    ULONG                                   nx_ip_fragment_failures;
    ULONG                                   nx_ip_total_fragments_sent;
    ULONG                                   nx_ip_total_fragments_received;
    ULONG                                   nx_ip_arp_requests_sent;
    ULONG                                   nx_ip_arp_requests_received;
    ULONG                                   nx_ip_arp_responses_sent;
    ULONG                                   nx_ip_arp_responses_received;
```

```
    ULONG                                   nx_ip_arp_aged_entries;
    ULONG                                   nx_ip_arp_invalid_messages;
    ULONG                                   nx_ip_arp_static_entries;
    ULONG                                   nx_ip_udp_packets_sent;
    ULONG                                   nx_ip_udp_bytes_sent;
    ULONG                                   nx_ip_udp_packets_received;
    ULONG                                   nx_ip_udp_bytes_received;
    ULONG                                   nx_ip_udp_invalid_packets;
    ULONG                                   nx_ip_udp_no_port_for_delivery;
    ULONG                                   nx_ip_udp_receive_packets_dropped;
    ULONG                                   nx_ip_udp_checksum_errors;
    ULONG                                   nx_ip_tcp_packets_sent;
    ULONG                                   nx_ip_tcp_bytes_sent;
    ULONG                                   nx_ip_tcp_packets_received;
    ULONG                                   nx_ip_tcp_bytes_received;
    ULONG                                   nx_ip_tcp_invalid_packets;
    ULONG                                   nx_ip_tcp_receive_packets_dropped;
    ULONG                                   nx_ip_tcp_checksum_errors;
    ULONG                                   nx_ip_tcp_connections;
    ULONG                                   nx_ip_tcp_passive_connections;
    ULONG                                   nx_ip_tcp_active_connections;
    ULONG                                   nx_ip_tcp_disconnections;
    ULONG                                   nx_ip_tcp_connections_dropped;
    ULONG                                   nx_ip_tcp_retransmit_packets;
    ULONG                                   nx_ip_tcp_resets_received;
    ULONG                                   nx_ip_tcp_resets_sent;
    ULONG                                   nx_ip_icmp_total_messages_received;
    ULONG                                   nx_ip_icmp_checksum_errors;
    ULONG                                   nx_ip_icmp_invalid_packets;
    ULONG                                   nx_ip_icmp_unhandled_messages;
    ULONG                                   nx_ip_pings_sent;
    ULONG                                   nx_ip_ping_timeouts;
    ULONG                                   nx_ip_ping_threads_suspended;
    ULONG                                   nx_ip_ping_responses_received;
    ULONG                                   nx_ip_pings_received;
    ULONG                                   nx_ip_pings_responded_to;
    ULONG                                   nx_ip_igmp_invalid_packets;
    ULONG                                   nx_ip_igmp_reports_sent;
    ULONG                                   nx_ip_igmp_queries_received;
    ULONG                                   nx_ip_igmp_checksum_errors;
    ULONG                                   nx_ip_igmp_groups_joined;
#ifndef NX_DISABLE_IGMPV2
    ULONG                                   nx_ip_igmp_router_version;
#endif
    ULONG                                   nx_ip_rarp_requests_sent;
    ULONG                                   nx_ip_rarp_responses_received;
    ULONG                                   nx_ip_rarp_invalid_messages;
    VOID  (*nx_ip_forward_packet_process)(struct NX_IP_STRUCT *, NX_PACKET *);
    ULONG                                   nx_ip_packet_id;
    struct NX_PACKET_POOL_STRUCT            *nx_ip_default_packet_pool;
    TX_MUTEX                                nx_ip_protection;
    UINT                                    nx_ip_initialize_done;
    NX_PACKET                               *nx_ip_driver_deferred_packet_head,
                                            *nx_ip_driver_deferred_packet_tail;
    VOID  (*nx_ip_driver_deferred_packet_handler)(struct NX_IP_STRUCT *, NX_PACKET *);
    NX_PACKET                               *nx_ip_deferred_received_packet_head,
                                            *nx_ip_deferred_received_packet_tail;
    VOID  (*nx_ip_raw_ip_processing)(struct NX_IP_STRUCT *, NX_PACKET *);
    NX_PACKET                               *nx_ip_raw_received_packet_head,
                                            *nx_ip_raw_received_packet_tail;
    ULONG                                   nx_ip_raw_received_packet_count;
    TX_THREAD                               *nx_ip_raw_packet_suspension_list;
    ULONG                                   nx_ip_raw_packet_suspended_count;
    TX_THREAD                               nx_ip_thread;
    TX_EVENT_FLAGS_GROUP                    nx_ip_events;
    TX_TIMER                                nx_ip_periodic_timer;
    VOID  (*nx_ip_fragment_processing)(struct NX_IP_DRIVER_STRUCT *);
    VOID  (*nx_ip_fragment_assembly)(struct NX_IP_STRUCT *);
    VOID  (*nx_ip_fragment_timeout_check)(struct NX_IP_STRUCT *);
    NX_PACKET                               *nx_ip_timeout_fragment;
```

```
    NX_PACKET                                   *nx_ip_received_fragment_head,
                                                *nx_ip_received_fragment_tail;
    NX_PACKET                                   *nx_ip_fragment_assembly_head,
                                                *nx_ip_fragment_assembly_tail;
    VOID  (*nx_ip_address_change_notify)(struct NX_IP_STRUCT *, VOID *);
    VOID                                        *nx_ip_address_change_notify_additional_info;
    ULONG                                       nx_ip_igmp_join_list[NX_MAX_MULTICAST_GROUPS];
    NX_INTERFACE                                *nx_ip_igmp_join_interface_list[NX_MAX_MULTICAST_GROUPS];
    ULONG                                       nx_ip_igmp_join_count[NX_MAX_MULTICAST_GROUPS];
    ULONG                                       nx_ip_igmp_update_time[NX_MAX_MULTICAST_GROUPS];
    UINT                                        nx_ip_igmp_group_loopback_enable[NX_MAX_MULTICAST_GROUPS];
    UINT                                        nx_ip_igmp_global_loopback_enable;
    void  (*nx_ip_igmp_packet_receive)(struct NX_IP_STRUCT *, struct NX_PACKET_STRUCT *);
    void  (*nx_ip_igmp_periodic_processing)(struct NX_IP_STRUCT *);
    void  (*nx_ip_igmp_queue_process)(struct NX_IP_STRUCT *);
    NX_PACKET                                   *nx_ip_igmp_queue_head;
    ULONG                                       nx_ip_icmp_sequence;
    void  (*nx_ip_icmp_packet_receive)(struct NX_IP_STRUCT *, struct NX_PACKET_STRUCT *);
    void  (*nx_ip_icmp_queue_process)(struct NX_IP_STRUCT *);
    NX_PACKET                                   *nx_ip_icmp_queue_head;
    TX_THREAD                                   *nx_ip_icmp_ping_suspension_list;
    ULONG                                       nx_ip_icmp_ping_suspended_count;
    struct NX_UDP_SOCKET_STRUCT                 *nx_ip_udp_port_table[NX_UDP_PORT_TABLE_SIZE];
    struct NX_UDP_SOCKET_STRUCT                 *nx_ip_udp_created_sockets_ptr;
    ULONG                                       nx_ip_udp_created_sockets_count;
    void   (*nx_ip_udp_packet_receive)(struct NX_IP_STRUCT *, struct NX_PACKET_STRUCT *);
    UINT                                        nx_ip_udp_port_search_start;
    struct NX_TCP_SOCKET_STRUCT                 *nx_ip_tcp_port_table[NX_TCP_PORT_TABLE_SIZE];
    struct NX_TCP_SOCKET_STRUCT                 *nx_ip_tcp_created_sockets_ptr;
    ULONG                                       nx_ip_tcp_created_sockets_count;
    void   (*nx_ip_tcp_packet_receive)(struct NX_IP_STRUCT *, struct NX_PACKET_STRUCT *);
    void   (*nx_ip_tcp_periodic_processing)(struct NX_IP_STRUCT *);
    void   (*nx_ip_tcp_fast_periodic_processing)(struct NX_IP_STRUCT *);
    void   (*nx_ip_tcp_queue_process)(struct NX_IP_STRUCT *);
    NX_PACKET                                   *nx_ip_tcp_queue_head,
                                                *nx_ip_tcp_queue_tail;
    ULONG                                       nx_ip_tcp_received_packet_count;
    struct NX_TCP_LISTEN_STRUCT                 nx_ip_tcp_server_listen_reqs[NX_MAX_LISTEN_REQUESTS];
    struct NX_TCP_LISTEN_STRUCT                 *nx_ip_tcp_available_listen_requests;
    struct NX_TCP_LISTEN_STRUCT                 *nx_ip_tcp_active_listen_requests;
    UINT                                        nx_ip_tcp_port_search_start;
    TX_TIMER                                    nx_ip_tcp_fast_periodic_timer;
    struct NX_ARP_STRUCT                        *nx_ip_arp_table[NX_ROUTE_TABLE_SIZE];
    struct NX_ARP_STRUCT                        *nx_ip_arp_static_list;
    struct NX_ARP_STRUCT                        *nx_ip_arp_dynamic_list;
    ULONG                                       nx_ip_arp_dynamic_active_count;
    NX_PACKET                                   *nx_ip_arp_deferred_received_packet_head,
                                                *nx_ip_arp_deferred_received_packet_tail;
    UINT   (*nx_ip_arp_allocate)(struct NX_IP_STRUCT *, struct NX_ARP_STRUCT **);
    void   (*nx_ip_arp_periodic_update)(struct NX_IP_STRUCT *);
    void   (*nx_ip_arp_queue_process)(struct NX_IP_STRUCT *);
    void   (*nx_ip_arp_packet_send)(struct NX_IP_STRUCT *,
                                                ULONG destination_ip), NX_INTERFACE *nx_interface);
    void   (*nx_ip_arp_gratuitous_response_handler)(struct NX_IP_STRUCT *, NX_PACKET *);
    void   (*nx_ip_arp_collision_notify_response_handler)(void *);
    void                                        *nx_ip_arp_collision_notify_parameter;
    ULONG                                       nx_ip_arp_collision_notify_ip_address;
    struct NX_ARP_STRUCT                        *nx_ip_arp_cache_memory;
    ULONG                                       nx_ip_arp_total_entries;
    void   (*nx_ip_rarp_periodic_update)(struct NX_IP_STRUCT *);
    void   (*nx_ip_rarp_queue_process)(struct NX_IP_STRUCT *);
    NX_PACKET                                   *nx_ip_rarp_deferred_received_packet_head,
                                                *nx_ip_rarp_deferred_received_packet_tail;
    struct NX_IP_STRUCT                         *nx_ip_created_next,
                                                *nx_ip_created_previous;
    void                                        *nx_ip_reserved_ptr;
    void   (*nx_tcp_deferred_cleanup_check)(struct NX_IP_STRUCT *);
    NX_INTERFACE nx_ip_interface[NX_MAX_IP_INTERFACES];
#ifndef NX_DISABLE_IP_STATIC_ROUTING
    NX_IP_ROUTING_ENTRY  nx_ip_routing_table[NX_IP_ROUTING_TABLE_SIZE];
    ULONG nx_ip_routing_table_entry_count;
#endif
```

```
} NX_IP;


typedef struct NX_IP_DRIVER_STRUCT
{
    UINT                                    nx_ip_driver_command;
    UINT                                    nx_ip_driver_status;
    ULONG                                   nx_ip_driver_physical_address_msw;
    ULONG                                   nx_ip_driver_physical_address_lsw;
    NX_PACKET                               *nx_ip_driver_packet;
    ULONG                                   *nx_ip_driver_return_ptr;
    struct NX_IP_STRUCT                     *nx_ip_driver_ptr;
} NX_IP_DRIVER;


typedef struct NX_IP_ROUTING_ENTRY_STRUCT
{
    ULONG                                   nx_ip_routing_entry_destination_ip;
    ULONG                                   nx_ip_routing_entry_net_mask;
    ULONG                                   nx_ip_routing_entry_next_hop_address;
    struct NX_INTERFACE_STRUCT              *nx_ip_routing_entry_ip_interface;
} NX_IP_ROUTING_ENTRY;


typedef  struct NX_PACKET_STRUCT
{
    struct NX_PACKET_POOL_STRUCT            *nx_packet_pool_owner;
    struct NX_PACKET_STRUCT                 *nx_packet_queue_next;
    struct NX_PACKET_STRUCT                 *nx_packet_tcp_queue_next;
    struct NX_PACKET_STRUCT                 *nx_packet_next;
    struct NX_PACKET_STRUCT                 *nx_packet_last;
    struct NX_PACKET_STRUCT                 *nx_packet_fragment_next;
    ULONG                                   nx_packet_length;
    struct NX_INTERFACE_STRUCT              *nx_packet_ip_interface;
    ULONG                                   nx_packet_next_hop_address
    UCHAR                                   *nx_packet_data_start;
    UCHAR                                   *nx_packet_data_end;
    UCHAR                                   *nx_packet_prepend_ptr;
    UCHAR                                   *nx_packet_append_ptr;
#ifdef NX_PACKET_HEADER_PAD
    ULONG                                   nx_packet_pad;
#endif
} NX_PACKET;


typedef struct NX_PACKET_POOL_STRUCT
{
    ULONG                                   nx_packet_pool_id;
    CHAR                                    *nx_packet_pool_name;
    ULONG                                   nx_packet_pool_available;
    ULONG                                   nx_packet_pool_total;
    ULONG                                   nx_packet_pool_empty_requests;
    ULONG                                   nx_packet_pool_empty_suspensions;
    ULONG                                   nx_packet_pool_invalid_releases;
    struct NX_PACKET_STRUCT                 *nx_packet_pool_available_list;
    CHAR                                    *nx_packet_pool_start;
    ULONG                                   nx_packet_pool_size;
    ULONG                                   nx_packet_pool_payload_size;
    TX_THREAD                               *nx_packet_pool_suspension_list;
    ULONG                                   nx_packet_pool_suspended_count;
    struct NX_PACKET_POOL_STRUCT            *nx_packet_pool_created_next,
                                            *nx_packet_pool_created_previous;
} NX_PACKET_POOL;


typedef struct NX_TCP_LISTEN_STRUCT
{
    UINT                                    nx_tcp_listen_port;
    VOID (*nx_tcp_listen_callback)(NX_TCP_SOCKET *socket_ptr, UINT port);
    NX_TCP_SOCKET                           *nx_tcp_listen_socket_ptr;
```

```
    ULONG                                 nx_tcp_listen_queue_maximum;
    ULONG                                 nx_tcp_listen_queue_current;
    NX_PACKET                             *nx_tcp_listen_queue_head,
                                          *nx_tcp_listen_queue_tail;
    struct NX_TCP_LISTEN_STRUCT           *nx_tcp_listen_next,
                                          *nx_tcp_listen_previous;
} NX_TCP_LISTEN;


typedef struct NX_TCP_SOCKET_STRUCT
{
    ULONG                                 nx_tcp_socket_id;
    CHAR                                  *nx_tcp_socket_name;
    UINT                                  nx_tcp_socket_client_type;
    UINT                                  nx_tcp_socket_port;
    ULONG                                 nx_tcp_socket_mss;
    ULONG                                 nx_tcp_socket_connect_ip;
    UINT                                  nx_tcp_socket_connect_port;
    ULONG                                 nx_tcp_socket_connect_mss;
    struct NX_INTERFACE_STRUCT            *nx_tcp_socket_connect_interface
    ULONG                                 nx_tcp_socket_next_hop_address
    ULONG                                 nx_tcp_socket_connect_mss2;
    ULONG                                 nx_tcp_socket_tx_slow_start_threshold;
    UINT                                  nx_tcp_socket_state;
    ULONG                                 nx_tcp_socket_tx_sequence;
    ULONG                                 nx_tcp_socket_rx_sequence;
    ULONG                                 nx_tcp_socket_rx_sequence_acked;
    ULONG                                 nx_tcp_socket_delayed_ack_timeout;
    ULONG                                 nx_tcp_socket_fin_sequence;
    ULONG                                 nx_tcp_socket_fin_received;
    ULONG                                 nx_tcp_socket_tx_window_advertised;
    ULONG                                 nx_tcp_socket_tx_window_congestion;
    ULONG                                 nx_tcp_socket_tx_outstanding_bytes;
    ULONG                                 nx_tcp_socket_ack_n_packet_counter;
    ULONG                                 nx_tcp_socket_rx_window_default;
    ULONG                                 nx_tcp_socket_rx_window_current;
    ULONG                                 nx_tcp_socket_rx_window_last_sent;
    ULONG                                 nx_tcp_socket_tx_window_current;
    ULONG                                 nx_tcp_socket_packets_sent;
    ULONG                                 nx_tcp_socket_bytes_sent;
    ULONG                                 nx_tcp_socket_packets_received;
    ULONG                                 nx_tcp_socket_bytes_received;
    ULONG                                 nx_tcp_socket_retransmit_packets;
    ULONG                                 nx_tcp_socket_checksum_errors;
    struct NX_IP_STRUCT                   *nx_tcp_socket_ip_ptr;
    ULONG                                 nx_tcp_socket_type_of_service;
    UINT                                  nx_tcp_socket_time_to_live;
    ULONG                                 nx_tcp_socket_fragment_enable;
    ULONG                                 nx_tcp_socket_receive_queue_count;
    NX_PACKET                             *nx_tcp_socket_receive_queue_head,
                                          *nx_tcp_socket_receive_queue_tail;
    ULONG                                 nx_tcp_socket_transmit_queue_maximum;
    ULONG                                 nx_tcp_socket_transmit_sent_count;
    NX_PACKET                             *nx_tcp_socket_transmit_sent_head,
                                          *nx_tcp_socket_transmit_sent_tail;
    ULONG                                 nx_tcp_socket_timeout;
    ULONG                                 nx_tcp_socket_timeout_rate;
    ULONG                                 nx_tcp_socket_timeout_retries;
    ULONG                                 nx_tcp_socket_timeout_max_retries;
    ULONG                                 nx_tcp_socket_timeout_shift;
    ULONG                                 nx_tcp_socket_keepalive_timeout;
    ULONG                                 nx_tcp_socket_keepalive_retries;
    struct NX_TCP_SOCKET_STRUCT           *nx_tcp_socket_bound_next,
                                          *nx_tcp_socket_bound_previous;
    TX_THREAD                             *nx_tcp_socket_bind_in_progress;
    TX_THREAD                             *nx_tcp_socket_receive_suspension_list;
    ULONG                                 nx_tcp_socket_receive_suspended_count;
    TX_THREAD                             *nx_tcp_socket_transmit_suspension_list;
    ULONG                                 nx_tcp_socket_transmit_suspended_count;
    TX_THREAD                             *nx_tcp_socket_connect_suspended_thread;
    TX_THREAD                             *nx_tcp_socket_disconnect_suspended_thread;
```

```
    TX_THREAD                               *nx_tcp_socket_bind_suspension_list;
    ULONG                                   nx_tcp_socket_bind_suspended_count;
    struct NX_TCP_SOCKET_STRUCT             *nx_tcp_socket_created_next,
                                            *nx_tcp_socket_created_previous;
    VOID (*nx_tcp_urgent_data_callback)(struct NX_TCP_SOCKET_STRUCT *socket_ptr);
    VOID (*nx_tcp_disconnect_callback)(struct NX_TCP_SOCKET_STRUCT *socket_ptr);
    VOID (*nx_tcp_receive_callback)(struct NX_TCP_SOCKET_STRUCT *socket_ptr);
    VOID                                    (*nx_tcp_socket_window_update_notify)
                                            (struct NX_TCP_SOCKET_STRUCT *socket_ptr);
    void                                    *nx_tcp_socket_reserved_ptr;
    ULONG                                   nx_tcp_socket_transmit_queue_maximum_default;
} NX_TCP_SOCKET;


typedef struct NX_UDP_SOCKET_STRUCT
{
    ULONG                                   nx_udp_socket_id;
    CHAR                                    *nx_udp_socket_name;
    UINT                                    nx_udp_socket_port;
    struct NX_IP_STRUCT                     *nx_udp_socket_ip_ptr;
    ULONG                                   nx_udp_socket_packets_sent;
    ULONG                                   nx_udp_socket_bytes_sent;
    ULONG                                   nx_udp_socket_packets_received;
    ULONG                                   nx_udp_socket_bytes_received;
    ULONG                                   nx_udp_socket_invalid_packets;
    ULONG                                   nx_udp_socket_packets_dropped;
    ULONG                                   nx_udp_socket_checksum_errors;
    ULONG                                   nx_udp_socket_type_of_service;
    UINT                                    nx_udp_socket_time_to_live;
    ULONG                                   nx_udp_socket_fragment_enable;
    UINT                                    nx_udp_socket_disable_checksum;
    ULONG                                   nx_udp_socket_receive_count;
    ULONG                                   nx_udp_socket_queue_maximum;
    NX_PACKET                               *nx_udp_socket_receive_head,
                                            *nx_udp_socket_receive_tail;
    struct NX_UDP_SOCKET_STRUCT             *nx_udp_socket_bound_next,
                                            *nx_udp_socket_bound_previous;
    TX_THREAD                               *nx_udp_socket_bind_in_progress;
    TX_THREAD                               *nx_udp_socket_receive_suspension_list;
    ULONG                                   nx_udp_socket_receive_suspended_count;
    TX_THREAD                               *nx_udp_socket_bind_suspension_list;
    ULONG                                   nx_udp_socket_bind_suspended_count;
    struct NX_UDP_SOCKET_STRUCT             *nx_udp_socket_created_next,
                                            *nx_udp_socket_created_previous;
    VOID (*nx_udp_receive_callback)(struct NX_UDP_SOCKET_STRUCT *socket_ptr);
    void                                    *nx_udp_socket_reserved_ptr;
    struct NX_INTERFACE_STRUCT              *nx_udp_socket_ip_interface;
} NX_UDP_SOCKET;
```

APPENDIX **D**

BSD-Compatible Socket API

BSD-Compatible Socket API

The BSD-Compatible Socket API supports a subset of the BSD Sockets API calls (with some limitations) by utilizing NetX® primitives underneath. This BSD-Compatible Sockets API layer should perform as fast or slightly faster than typical BSD implementations because this API utilizes internal NetX primitives and bypasses unnecessary NetX error checking.

Configurable options allow the host application to define the maximum number of sockets, TCPmaximum window size, and depth of listen queue.

Due to performance and architecture issues, this BSD-Compatible Sockets API does not support all BSD Sockets calls. In addition, not all BSD options are available for the BSD services, specifically the following:

- ***select***() call works with only fd_set *readfds, other arguments in this call e.g., writefds, exceptfds are not supported.
- INT flags argument is not supported for ***send()***, ***recv()***, ***sendto(),*** and ***recvfrom ()*** calls.
- The BSD-Compatible Socket API supports only limited set of BSD Sockets calls.

The source code is designed for simplicity and is comprised of only two files, ***nx_bsd.c*** and ***nx_bsd.h***. Installation requires adding these two files to the build project (not the NetX library) and creating the host application which will use BSD Socket service calls. The ***nx_bsd.h*** file must also be included in your application source. Sample demo files are included with the distribution which is freely available with NetX. Further details are available in the help and Readme files bundled with the BSD-Compatible Socket API package.

The BSD-Compatible Sockets API supports the following BSD Sockets API calls:

INT ***getpeername****(INT sockID, struct sockaddr *remoteAddress, INT *addressLength);*

INT ***getsockname****(INT sockID, struct sockaddr *localAddress, INT *addressLength);*

INT ***recvfrom****(INT sockID, CHAR *buffer, INT buffersize, INT flags,struct sockaddr *fromAddr, INT *fromAddrLen);*

INT ***recv****(INT sockID, VOID *rcvBuffer, INT bufferLength, INT flags);*

INT ***sendto****(INT sockID, CHAR *msg, INT msgLength, INT flags, struct sockaddr *destAddr, INT destAddrLen);*

INT ***send****(INT sockID, const CHAR *msg, INT msgLength, INT flags);*

INT ***accept****(INT sockID, struct sockaddr *ClientAddress, INT *addressLength);*

INT ***listen****(INT sockID, INT backlog);*

INT ***bind*** *(INT sockID, struct sockaddr *localAddress, INT addressLength);*

INT ***connect****(INT sockID, struct sockaddr *remoteAddress, INT addressLength);*

INT ***socket****(INT protocolFamily, INT type, INT protocol);*

INT ***soc_close*** *(INT sockID);*

INT ***select****(INT nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);*

VOID ***FD_SET****(INT fd, fd_set *fdset);*

VOID ***FD_CLR****(INT fd, fd_set *fdset);*

INT ***FD_ISSET****(INT fd, fd_set *fdset);*

VOID ***FD_ZERO****(fd_set *fdset);*

APPENDIX **E**

ASCII Character Codes

ASCII Character Codes in HEX

most significant nibble

least signigicant nibble	0_	1_	2_	3_	4_	5_	6_	7_
_0	NUL	DLE	SP	0	@	P	'	p
_1	SOH	DC1	!	1	A	Q	a	q
_2	STX	DC2	"	2	B	R	b	r
_3	ETX	DC3	#	3	C	S	c	s
_4	EOT	DC4	$	4	D	T	d	t
_5	ENQ	NAK	%	5	E	U	e	u
_6	ACK	SYN	&	6	F	V	f	v
_7	BEL	ETB	'	7	G	W	g	w
_8	BS	CAN	(	8	H	X	h	x
_9	HT	EM	)	9	I	Y	i	y
_A	LF	SUB	*	:	J	Z	j	z
_B	VT	ESC	+	;	K	[	K	}
_C	FF	FS	,	<	L	\	l	|
_D	CR	GS	-	=	M	]	m	}
_E	SO	RS	.	>	N	^	n	~
_F	SI	US	/	?	O	_	o	DEL

Index

Symbols

_nx_arp_packet_deferred_receive 47, 359, 361
_nx_ip_driver_deferred_processing 48, 356
_nx_ip_packet_deferred_receive 48, 359
_nx_ip_packet_receive 47, 359, 361
_nx_ip_thread_entry 46
_nx_rarp_packet_deferred_receive 47, 359, 361
_nx_version_id 39

Numerics

16-bit checksum 86, 93
16-bit identification 62
48-bit address support 71

A

accelerated software development process 20
accepting a TCP server connection 250
access functions 47
ACK 53
    returned 101
adding deferred packet logic to the NetX IP helper thread 360
adding static route 198
Address Resolution Protocol (see ARP) 71
address specifications
    broadcast 60
    multicast 60
    unicast 60
all hosts address 81
allocating a packet from specified pool 204
allocating memory packets 49
allocation of packets for ARP, RARP, ICMP, IGMP 67
ANSI C 17, 21
appending data to end of packet 208
application development area 26
application interface calls 45
application migration path 21
application source and link 27
application specific modifications 17
application threads 27, 44
ARP 27
    periodic processing 46
    processing 72
ARP aging 74
    disabled 75
ARP cache 72
ARP dynamic entries 72
ARP Enable 72
ARP enable service 72
ARP entry from dynamic ARP entry list 72
ARP entry setup 72
ARP information gathering
    disabling 30
ARP messages 73
    Ethernet destination address 73
    Ethernet source address 73
    frame type 74
    hardware size 74
    hardware type 74
    operation code 74
    protocol size 74
    protocol type 74
    sender Ethernet address 74
    sender IP address 74
    target Ethernet address 74
    target IP address 74

ARP packets 28
format 76
processing 46
ARP request information in the ARP cache 30
ARP request message 73
ARP requests 72, 73
ARP response 73
ARP response send 350
ARP response send request 350
ARP send 349
ARP send packet request 350
ARP static entries 72
ARP statistics and errors 75
ARP timeout 47
array of internal ARP mapping data structures 72
ASCII
character codes in HEX 416
format 24
asynchronous events 47
attach interface 352
attach interface request 352
attaching network interface to IP instance 182
automatic invalidation of dynamic ARP entries 74

B

backing up NetX distribution disk 26
basic structure for NetX physical media drivers 361
big endian 61, 74, 77, 79, 84, 85, 93
binary version of NetX C library 26
binding client TCP socket to TCP port 234
binding UDP socket to UDP port 314
black box 17
broadcast addresses 60
BSD-compatible socket API 18
buffering packets 47
building a NetX application 27
building a TCP header 100
building a valid NetX packet 359
building the NetX runtime library 26
bypassing changes to see if problem changes 28
byte swapping on little endian environments 30

C

C compilers 11
C header files 25
C include files 26
calculation of capacity of pool 55
callback function 46
calling thread's context 45
causing IP instance to leave specified multicast group 150
chaining fixed size packet structures 55
characteristics of IP instance found in control block 68
characteristics of packet memory pool 58
checking status of an IP instance 202
checking status of attached IP interface 186
checksum 35, 45
checksum calculation 100
checksum logic
disabling 35
checksum logic on IP packets sent
disabling 32
checksum logic on received TCP packets
disabling 35
checksum processing in lower-priority threads 45
Class D IP address 81
Class D multicast addresses 82
classes of IP addresses 58
client binding 102
client connection requests 100
commercial network stacks 17
compatibility with legacy NetX Ethernet drivers 65
compilation 27
complete NetX source code 25
complex protocols 48
configuration 28

configuring socket's transmit parameters 302
connecting a client TCP socket 236
connection events 94
connection management 49
connection request to a TCP server 95
connection service 102
connectionless protocols 60
connectionless sending and receiving of data 49
contents of distribution disk 25
copying packet 206
CRC logic 32
CRC processing 35
create services 45
creating a packet pool in specified memory area 216
creating a static IP to hardware mapping in ARP cache 128
creating a TCP client or server socket 272
creating a UDP socket 322
creating an IP instance 158
creating IP instance with IP address of zero 76
creating IP instances 66
Customer Support Center 12

D

data encapsulation 49
data structures 25
data transfer between network members 85
datagram
definition 62
debug packet dumping 29
debugging 24
default packet pool 44, 66, 67
deferred driver packet handling 29
deferred IP packet reception 46
deferred processing event 357
deferred processing queue 48
deferred receive packet handling 360
deferring interrupt processing 48
deleting a previously created IP instance 160
deleting a previously created packet pool 218
deleting a static IP to hardware mapping in the ARP cache 130
deleting a TCP socket 276
deleting a UDP socket 324
deleting all static ARP entries 126
deleting static route 200
delivering packet to first suspended thread 101
demo application 25, 27
demo_netx.c 25
demonstration system 27
destination address of the packet 71
destination IP address 64
development investment protection 21
development process 20
development tool environment 26
development tool options 38
development tool specific data definitions 25
development tools 25
disabling checksum for the UDP socket 318
disabling error checking 29
disabling IGMP loopback 142
disabling IP packet forwarding 164
disabling IP packet fragmenting 168
disabling link 347
disabling listening for client connection on TCP port 266
disabling raw packet sending/receiving 188
disabling Reverse Address Resolution Protocol (RARP) 226
disabling the UDP checksum logic 87
disconnect callbacks 46
disconnect processing 99
disconnecting client and server socket connections 278
disconnection services 102
distribution disk 38
distribution disk contents 25
driver deferred processing 356

driver entry 345
driver initialization 46, 67, 346, 360
driver input 359
driver introduction 344
driver output 358
driver output function 348
driver request data structure 345
driver requests 345
DSP 16
duplex type request 354
dynamic ARP entries 72
dynamically mapping 32-bit IP addresses 71

E

ease of use 20
easy-to-use interface 20
embedded development 24
embedded network applications 19
enable link 347
enable services 45
enabling Address Resolution Protocol (ARP) 116
enabling checksum for the UDP socket 320
enabling ICMP processing 78
enabling IGMP loopback 144
enabling Internet Control Message Protocol (ICMP) component 132
enabling Internet Group Management Protocol (IGMP) component 138
enabling IP packet forwarding 166
enabling IP packet fragmenting 170
enabling listening for client connection on TCP port 254
enabling raw packet sending/receiving 190
enabling Reverse Address Resolution Protocol (RARP) 228
enabling TCP component of NetX 242
enabling UDP component of NetX 306
ensuring driver supplies ARP and IP packets 28
entry point of internal IP thread 46
Ethernet 71
Ethernet ARP requests formats 73
examining default packet pool 28
examining NX_IP structure 28
external ping request 80
extracting data from packet via an offset 210
extracting IP and sending port from UDP datagram 340
extracting network parameters from UDP packet 312

F

fast response 19
files common to product distributions 25
finding next available TCP port 244
finding next available UDP port 308
first packet of next separate packet 53
fixed-size memory blocks 50
fixed-size packet pools 50
flow control for data transfer 100
fragment timeout processing 47
fragmentation 50
fragmented IP packets 65
freeing up processor cycles 16
function prototype for registration of the deferred packet handler 360
function prototype for the deferred receive function 360
functional components of NetX 41

G

gateway IP address 60
getting allocation errors 356
getting duplex type 354
getting error count 354
getting length of packet data 214
getting link speed 353
getting link status 353
getting MSS of socket 284
getting MSS of socket peer 286
getting port number bound to client TCP socket 238
getting receive packet count 355
getting transmit packet count 355

global data structures 25
guide conventions 10

H

handling
periodic processing 67
handling connection and disconnection actions 101
handling deferred packet processing 67
head and tail pointers of the transmit queue 358
headers 49
headers in the TCP/IP implementation 61, 93
higher-level protocols 61
high-speed Internet connectivity 19
host system considerations 24
hosts with multiple interfaces 65

I

I/O 44
IBM-PC hosts 24
ICMP 48
ICMP debug log 31
ICMP enable 78
ICMP header format 79
ICMP information gathering
disabling 31
ICMP ping message format 79
ICMP statistics and errors 80
IGMP 48
IGMP debug log 32
IGMP enable 81
IGMP header 83
IGMP header format 83
IGMP initialization 81
IGMP periodic processing 46
IGMP processing 46, 81
IGMP query messages 84
format 84
IGMP report 82
IGMP report message 83
IGMP report message format 83
IGMP statistics and errors 84
IGMP timeout 47
IGMP v2 support disabled 32
image download to target 27
implemented as a C library 16
improved responsiveness 19
incoming IP packets 47, 54
increased throughput 20
increasing stack size during the IP create 67
initial execution 44
initial IP address 66
initialization 44, 45, 66, 102
NetX system 27
of driver 46
initializing NetX system 232
initiating ARP requests 64
in-line processing 16
instruction image requirements 16
integrated with 21
interface and next hop address 94
interface control block 66
interface control block assigned to the packet 54
internal component function calls 16
internal IP send processing 348
internal IP send routine 100
Internal IP thread 45
internal IP thread 44, 46, 48, 65
internal IP thread calls 46
internal transmit sent queue 101
Internet Control Message Protocol (see ICMP) 78
Internet Group Management Protocol (see IGMP) 81, 377
Internet Protocol (see IP) 58
interrupt routine 48
interrupt service routine 47
invalidating all dynamic entries in ARP cache 112
IP address 58, 68
13-bit fragment offset 63
16-bit checksum 63
16-bit total length 62
32-bit destination IP address 63

32-bit source IP address 63
3-bit flags 62
4-bit header length 62
4-bit version 62
8-bit protocol 63
8-bit time to live (TTL) 63
8-bit type of service (TOS) 62
gateway 60
IP address structure 59
IP checksum 64
logic 32
IP control block
NX_IP 68
IP create call 46
IP create service 67
IP data structure 44
IP datagram 62
IP debug log 33
IP fragment assembly timeouts 46
IP fragmentation 63
disabling 64
IP fragmentation logic
disabling 32
IP header 60, 64
IP header format 61
IP helper thread 67, 101, 347
IP information gathering
disabling 32
IP instance 27, 67
control blocks 68
creation 44
IP packet fragment assembly 46
IP packets 28, 47, 48
IP periodic timers 47
IP pseudo header 86, 93
IP receive 65
IP receive processing 65
IP resources 44
IP send 64
IP send function 45
IP send processing 64
IP statistics and errors 68
IP version 4 62
IP_ADDRESS 60
ISR processing time 48
issuing a command to the network driver 162

J

joining IP interface to specified multicast group 146
joining the specified multicast group 148

L

last packet within the same network packet 53
layering 49
least significant 32-bits of physical address 348, 350
least significant 32-bits of physical multicast address 351, 352
line speed request 353
link 27
link allocation error count request 356
link enable call 46
link error count request 355
link level 48
link receive packet count request 355
link status request 353
link transmit packet count request 356
linked-list manipulation 50
linked-list processing 50
Linus (Unix) development platforms 26
listen callbacks 46
listening for packets with the Ethernet address 82
locating a physical hardware address given an IP address 120
locating an IP address given a physical address 124
logical connection point in the TCP protocol 94
logical loopback interface 69
long-word alignment 55
long-word boundary 55
loopback packet out the logical interface 70
low packet overhead path 87
lowest layer protocol 48

M

maintaining relationship between IP address and physical hardware address 72
management
Internet Control Message Protocol (ICMP) 376
Internet Protocol (IP) 377
Reverse Address Resolution Protocol (RARP) 379
Transmission Control Protocol (TCP) 379
management-type protocols 48
managing the flow of data 48
maximum number of ARP retries without ARP response 31
maximum number of multicast groups that can be joined 32
media driver 66
memory and priority of the internal IP thread 66
memory areas
NetX objects 57
ThreadX 57
microprocessors 19
minimal source code 25
minimizing dropped packets 360
minimizing ISR processing 360
most significant 32-bits of physical address 348, 350
most significant 32-bits of physical multicast address 351, 352
multicast addresses 60
multicast group 81
multicast group join 82, 351
multicast group join request 351
multicast group leave 82, 351
multicast group leave request 352
multicast groups on the primary network 82
multicast IP addresses 81
multicast routers 84
multihome devices 94
multihome host 70
multihome host application 66
multihome hosts 46, 47, 75, 77, 82, 95, 361
multihome support 69
multihome support service
nx_igmp_multicast_interface_join 70
nx_ip_interface_address_get 70
nx_ip_interface_address_set 70
nx_ip_interface_attach 70
nx_ip_interface_info_get 70
nx_ip_interface_status_check 70
nx_ip_raw_packet_interface_send 70
nx_udp_socket_interface_send 70
multiple interfaces 70
multiple linked lists 72
multiple network interface support 69
multiple physical network interfaces 69
multiple pools of fixed-size network packets 50
multiple thread suspension 57

N

naming convention 26
network data packets 49
network destination IP addresses 70
network driver 17, 44, 46, 47, 48, 63, 64
network hardware 20
network layer 48
network mask 66, 68
network packets on a queue 55
network stack 17
network traffic 20
NetX architecture 20
NetX ARP software 75
NetX benefits 19
NetX callback functions 46
NetX constants 383
alphabetic listings 384
NetX data structures 27
NetX data types 403
NetX distribution 26
NetX error checking API removal 29
NetX IGMP software 84
NetX installation 26

NetX IP send routine 17
NetX IP software 68
NetX library 27
NetX library file 26
NetX packet management software 57
NetX packet pool create 56
NetX physical media drivers 343
NetX port 25, 27
NetX protocol stack 19, 25
NetX RARP software 78
NetX runtime library 27
NetX services 27, 105, 375
NetX source code 24
NetX system initialization 27
NetX unique features 16
NetX Version ID 39
netx.txt 39
new application threads 28
new processor architecture 21
next packet within same network packet 53
notifying application if IP address changes 152
notifying application of each received packet 334
notifying application of received packets 294
notifying application of window size updates 304
number of bytes in entire network packet 54
number of bytes in the memory area 55
number of keepalive retries before connection is broken 36
number of packets queued while waiting for an ARP response 31
number of seconds ARP entries remain valid 31
number of seconds between ARP retries 31
number of ThreadX timer ticks in one second 33
nx_api.h 25, 26, 27, 31, 32, 33, 35, 36, 58, 60, 68, 90, 103
NX_ARP_DISABLE_AUTO_ARP_ENTRY 30
nx_arp_dynamic_entries_invalidate 112
nx_arp_dynamic_entry_set 114
nx_arp_enable 72, 116
NX_ARP_EXPIRATION_RATE 31, 75
nx_arp_gratuitous_send 118
nx_arp_hardware_address_find 120
nx_arp_info_get 75, 122
nx_arp_ip_address_find 124
NX_ARP_MAX_QUEUE_DEPTH 31, 64
NX_ARP_MAXIMUM_RETRIES 31, 73
nx_arp_static_entries_delete 126
nx_arp_static_entry_create 72, 128
nx_arp_static_entry_delete 130
NX_ARP_UPDATE_RATE 31, 73
NX_DEBUG 29
NX_DEBUG_PACKET 29
NX_DISABLE_ARP_INFO 30
NX_DISABLE_ERROR_CHECKING 29
NX_DISABLE_FRAGMENTATION 32, 64
NX_DISABLE_ICMP_INFO 31
NX_DISABLE_IGMP_INFO 32
NX_DISABLE_IGMPV2 32
NX_DISABLE_IP_INFO 32
NX_DISABLE_IP_RX_CHECKSUM 32
NX_DISABLE_IP_TX_CHECKSUM 32
NX_DISABLE_LOOPBACK_INTERFACE 33, 69, 70
NX_DISABLE_PACKET_INFO 34
NX_DISABLE_RARP_INFO 34
NX_DISABLE_RESET_DISCONNECT 34
NX_DISABLE_RX_SIZE_CHECKING 33
NX_DISABLE_TCP_INFO 34
NX_DISABLE_TCP_RX_CHECKSUM 35
NX_DISABLE_TCP_TX_CHECKSUM 35
NX_DISABLE_UDP_INFO 38
NX_DRIVER_DEFERRED_PROCESSING 29, 360
NX_ENABLE_IP_STATIC_ROUTING 33, 71
nx_icmp_enable 78, 132
NX_ICMP_ENABLE_DEBUG_LOG 31
nx_icmp_info_get 81, 134
nx_icmp_ping 136
nx_igmp_enable 81, 138
NX_IGMP_ENABLE_DEBUG_LOG 32

nx_igmp_info_get 84, 140
nx_igmp_loopback_disable 142
nx_igmp_loopback_enable 144
nx_igmp_multicast_interface_join 82, 146, 351
nx_igmp_multicast_join 82, 148, 351
nx_igmp_multicast_leave 82, 150, 351
NX_INTERFACE 344, 347, 357
nx_interface_additional_link_info 357
nx_interface_ip_mtu_size 63
nx_interface_link_up 347, 353
NX_IP structure 344
nx_ip_address_change_notifiy 152
nx_ip_address_get 152, 154
nx_ip_address_set 156
nx_ip_create 45, 47, 66, 67, 69, 71, 75, 158, 346
nx_ip_delete 160, 347
NX_IP_DRIVER 345, 346, 347, 348, 349, 350, 352, 353, 355, 356, 357
nx_ip_driver_commad 345
nx_ip_driver_command 346, 347, 348, 349, 350, 351, 352, 353, 354, 355, 356, 357
nx_ip_driver_direct_command 162, 353, 354, 355, 356, 357
nx_ip_driver_interface 346, 347, 348, 349, 350, 351, 352, 353, 354, 355, 356, 357
nx_ip_driver_packet 348, 349, 350
nx_ip_driver_physical_address_lsw 348, 349, 350, 351, 352
nx_ip_driver_physical_address_msw 348, 349, 350, 351, 352
nx_ip_driver_ptr 346, 347, 348, 349, 350, 351, 352, 353, 354, 355, 356, 357
nx_ip_driver_return_ptr 353, 354, 355, 356, 357
nx_ip_driver_status 345, 352
NX_IP_ENABLE_DEBUG_LOG 33
nx_ip_forwarding_disable 164
nx_ip_forwarding_enable 166
nx_ip_fragment_disable 168
nx_ip_fragment_enable 170
nx_ip_gateway_address_set 172
nx_ip_info_get 68, 174
nx_ip_interface 54
nx_ip_interface_address_get 178
nx_ip_interface_address_set 180
nx_ip_interface_attach 66, 69, 75, 182, 346, 352
nx_ip_interface_info_get 184
nx_ip_interface_status_check 77, 186, 353
NX_IP_PERIODIC_RATE 33, 35
nx_ip_raw_packet_disable 188
nx_ip_raw_packet_enable 190
nx_ip_raw_packet_enabled 65
nx_ip_raw_packet_interface_send 65, 192
nx_ip_raw_packet_receive 66, 194
nx_ip_raw_packet_send 65, 196
NX_IP_ROUTING_TABLE_SIZE 33
nx_ip_socket_send 87
nx_ip_static_route_add 71, 198
nx_ip_static_route_delete 71, 200
nx_ip_status_check 45, 77, 202, 353
NX_LINK_ARP_RESPONSE_SEND 350
NX_LINK_ARP_SEND 349
NX_LINK_DISABLE 348
NX_LINK_ENABLE 347
NX_LINK_GET_ALLOC_ERRORS 356
NX_LINK_GET_DUPLEX_TYPE 354
NX_LINK_GET_ERROR_COUNT 355
NX_LINK_GET_RX_COUNT 355
NX_LINK_GET_SPEED 353
NX_LINK_GET_STATUS 353
NX_LINK_GET_TX_COUNT 356
NX_LINK_INITIALIZE 346
NX_LINK_MULTICAST_JOIN 351
NX_LINK_MULTICAST_LEAVE 352
NX_LINK_PACKET_BROADCAST 349
NX_LINK_PACKET_SEND 348
NX_LINK_RARP_SEND 350
NX_LINK_USER_COMMAND 357
NX_LITTLE_ENDIAN 30
NX_MAX_IP_INTERFACES 33, 69
NX_MAX_LISTEN_REQUESTS 35
NX_MAX_MULTICAST_GROUPS 32

NX_MAX_PHYSICAL_INTERFACES 30, 69
nx_next_hop_address 54
NX_PACKET 54, 55, 101
nx_packet_allocate 204
NX_PACKET_ALLOCATED 53
nx_packet_append_ptr 55
nx_packet_copy 206
nx_packet_data_append 208
nx_packet_data_end 54
nx_packet_data_extract_offset 210
nx_packet_data_retrieve 212
nx_packet_data_start 54
NX_PACKET_ENABLE_DEBUG_LOG 34
nx_packet_fragment_next 54
NX_PACKET_FREE 53
nx_packet_last 53
nx_packet_length 54
nx_packet_length_get 214
nx_packet_next 53
nx_packet_pool_create 54, 216
nx_packet_pool_delete 218
nx_packet_pool_info_get 58, 220
nx_packet_prepend_ptr 55
nx_packet_queue_next 53, 358
nx_packet_release 222
nx_packet_tcp_queue_next 53
nx_packet_transmit_release 53, 224, 358
NX_PHYSICAL_HEADER 30
NX_PHYSICAL_TRAILER 30
nx_port.h 11, 25, 26, 33
nx_rarp_disable 226
nx_rarp_enable 228
NX_RARP_ENABLE_DEBUG_LOG 34
nx_rarp_info_get 78, 230
NX_RARP_UPDATE_RATE 77
nx_rnd.c 361
NX_SUCCESS 345
nx_system_initialize 27, 44, 232
NX_TCP_ACK_EVERY_N_PACKETS 35
NX_TCP_ACK_TIMER_RATE 35
nx_tcp_client_socket_bind 95, 234
nx_tcp_client_socket_connect 95, 236
nx_tcp_client_socket_port_get 238
nx_tcp_client_socket_unbind 97, 240
nx_tcp_enable 91, 242
NX_TCP_ENABLE_DEBUG_LOG 35
NX_TCP_ENABLE_KEEPALIVE 35
NX_TCP_FAST_TIMER_RATE 36
nx_tcp_free_port_find 244
NX_TCP_IMMEDIATE_ACK 36
nx_tcp_info_get 103, 246
NX_TCP_KEEPALIVE_INITIAL 36
NX_TCP_KEEPALIVE_RETRIES 36
NX_TCP_KEEPALIVE_RETRY 36
NX_TCP_MAXIMUM_RETRIES 37
NX_TCP_MAXIMUM_TX_QUEUE 37
NX_TCP_RETRY_SHIFT 38
nx_tcp_server_socket_accept 98, 99, 250
nx_tcp_server_socket_listen 98, 100, 254
nx_tcp_server_socket_relisten 98, 100, 258
nx_tcp_server_socket_unaccept 99, 262
nx_tcp_server_socket_unlisten 100, 266
NX_TCP_SOCKET 103
nx_tcp_socket_bytes_available 270
nx_tcp_socket_create 95, 97, 102, 272
nx_tcp_socket_delete 97, 276
nx_tcp_socket_disconnect 95, 99, 278
nx_tcp_socket_info_get 103, 280
nx_tcp_socket_mss_get 284
nx_tcp_socket_mss_peer_get 286
nx_tcp_socket_mss_set 288
nx_tcp_socket_peer_info_get 290
nx_tcp_socket_receive 292
nx_tcp_socket_receive_notify 101, 294
nx_tcp_socket_send 100, 296
nx_tcp_socket_state_wait 300
nx_tcp_socket_transmit_configure 302
nx_tcp_socket_window_update_notify 304
NX_TCP_TRANSMIT_TIMER_RATE 38
nx_tcp.h 35, 36, 38
nx_udp_enable 85, 306
NX_UDP_ENABLE_DEBUG_LOG 38
nx_udp_free_port_find 308
nx_udp_info_get 90, 310
nx_udp_packet_info_extract 312
NX_UDP_SOCKET 90
nx_udp_socket_bind 314
nx_udp_socket_bytes_available 316

nx_udp_socket_checksum_disable 87, 318
nx_udp_socket_checksum_enable 320
nx_udp_socket_create 89, 322
nx_udp_socket_delete 324
nx_udp_socket_info_get 90, 326
nx_udp_socket_interface_send 328
nx_udp_socket_port_get 330
nx_udp_socket_receieve_notify 89
nx_udp_socket_receive 45, 88, 332
nx_udp_socket_receive_notify 334
nx_udp_socket_send 17, 45, 87, 336
nx_udp_socket_unbind 338
nx_udp_source_extract 340
NX_UNHANDLED_COMMAND 358
nx.a (or nx.lib) 27
nx.lib 26
nxd_udp_socket_extract 146, 193

O

optimal packet payload size 50
outgoing fragmentation 64
overwriting memory
IP helper thread 67

P

packet allocation 50
packet broadcast 348
packet broadcast request 349
packet deallocation 50
packet destination IP address 70
packet header 56
packet interface 70
packet memory pool 57
packet memory pools 49
packet payload 57
packet pool control block
NX_PACKET_POOL 58
packet pool control blocks 58
packet pool creation 44
packet pool information gathering
disabling 34
packet pool memory area 57
packet pools 50
packet reception 47
packet send 348
packet send processing 348
packet size 55
packet transmission 47, 53
packet transmission completion 47
packet_ptr 361
packet-receive processing 17
packets
fast processing of 20
packets requiring IP address resolution 64
partitioning network aspect 20
passing error and control information
between IP network members 78
path for the development tools 26
payload size 54, 55
payload size for packets in pool 58
performance advantages 16
periodic RARP request 77
periodic timers 47
physical address mapping 82
physical Ethernet addresses 82
physical layer header removed 359
physical media 71
physical media driver 53, 57
physical media header 348
physical packet header size 30
picking up port number bound to UDP
socket 330
Piconet™ architecture 16
ping processing 46
ping request 78
ping response 79, 80
ping response message 80
placing a raw packet on an IP instance 29
placing packets with receive data on TCP
socket receive queue 101
pointer to IP instance 346, 347, 348, 349, 350, 351, 352, 353, 354, 355, 356, 357
pointer to the destination to place the
allocation error count 356
pointer to the destination to place the
duplex type 354

pointer to the destination to place the error count 355
pointer to the destination to place the line speed 354
pointer to the destination to place the receive packet count 355
pointer to the destination to place the status 353
pointer to the destination to place the transmit packet count 356
Pointer to the packet to send 349
pointer to the packet to send 348, 349, 350
pointer to the physical network interface 346, 347, 348, 349, 350, 351, 352, 353, 354, 355, 356, 357
pool capacity 55
pool statistics and errors 57
portability 11, 17
pre-defined multicast addresses 81
preemption 46
premium NetX 25
premium package 26
prepend pointer 65
prepend pointer in the packet 64
prevention of stalling network requests 45
primary interface 69
print debug information 29
priority and stack size of internal IP thread 45
processing needs 25
processing packet and periodic requests 67
processing requirements on a single packet 20
processor isolation 20
processor-independent interface 20
product distribution 25
program execution overview 44
protecting software investment 21
public domain network stacks 18

Q

queue network packets 53
queue packets 53
queued client connection request packets 100

R

RAM driver example 361
RARP debug log 34
RARP enable 76
RARP information gathering disabling 34
RARP packets 47
RARP reply 77
RARP reply messages 77
RARP reply packet 77
RARP request 76
RARP request packet format 76
RARP send 350
RARP send request 350
RARP statistics and errors 78
raw IP packet processing 66
raw IP packets 65
raw IP receive 66
raw IP send 65
readme_netx.txt 24, 25, 26, 27, 29, 38
ready-to-execute mode 45
real-time network software 19
receive functions 361
receive packet callback 89
receive packet callback function 101
receive packet dispatched 359
receive packet interrupt processing 56
receive packet processing 360
received packet 45
receiving a raw IP packet 194
receiving data from a TCP socket 292
receiving datagram from UDP socket 332
recompiling NetX library with debug options 28
recompiling the NetX library 28
releasing a previously allocated packet 222
releasing a transmitted packet 224
reliable data path 49
re-listening for client connection on TCP port 258

removing association between server socket and server port 262
retransmit timeout period changes between retries 38
retrieving data from packet 212
retrieving information
ARP activities 122
ICMP activities 134
IGMP activities 140
retrieving information about IP activities 174
retrieving information about packet pool 220
retrieving information about peer TCP socket 290
retrieving information about RARP activities 230
retrieving information about TCP activities 246
retrieving information about TCP socket activities 280
retrieving information about UDP activities 310
retrieving information about UDP socket activities 326
retrieving interface IP address 178
retrieving network interface parameters 184
retrieving number of bytes available for retrieval 270, 316
retrieving the IP address and network mask 154
Reverse Address Resolution Protocol (see RARP) 75
RFC
1112 81
768 85
791 58
792 78
793 91
826 71
RFC 903 75
RFCs Supported by NetX
RFC 1112 18
RFC 2236 18
RFC 768 18
RFC 791 18
RFC 792 18
RFC 793 18
RFC 826 18
RFC 903 18
RISC 16
runtime by application threads 66
runtime image 16

S

scaling 16
seconds between retries of the keepalive timer 36
seconds of inactivity before the keepalive timer activates
defining 36
send packet request 348, 350
sending a raw IP packet 196
sending a UDP packet 85
sending and receiving of data 49
sending and receiving simple packets 48
sending and receiving TCP, UDP, ICMP, and IGMP messages 58
sending and receiving UDP packets 85
sending data through a TCP socket 296
sending datagram through UDP socket 328, 336
sending gratuitous ARP request 118
sending or receiving UDP data 87
sending ping request to specified IP address 136
sending raw IP packet out specified network interface 192
sending request to unmapped IP address 72
server listen requests
defining 35
service call data type 11
CHAR 11
UINT 11
ULONG 11
VOID 11
service call interface 11, 20

service prototypes 25
setting dynamic ARP entry 114
setting Gateway IP address 172
setting interface IP address and network mask 180
setting MSS of socket 288
setting the IP address and network mask 156
setup and data transfer phase of a connection 100
simulated network driver 27
size in bytes of the physical packet trailer 30
size of
NetX 16
socket interface 70
socket output queue 53
socket receive function 88
socket receive queue 88
socket transmit queue 99
socket waiting for a connection 100
sockets 27
software maintenance 20
source code
ANSI C 17
specification of IP addresses 60
stack sizes 28
standard NetX 25
standard package 26
start of the physical payload area 54
static ARP mapping 72
static IP routing 70
static routing table 71
statistics 57
free packets in pool 57
invalid packet releases 58
pool empty allocation requests 58
pool empty allocation suspensions 58
TCP socket bytes received 102
TCP socket bytes sent 102
TCP socket checksum errors 103
TCP socket packet retransmits 103
TCP socket packets queued 103
TCP socket packets received 102
TCP socket packets sent 102
TCP socket receive window size 103
TCP socket state 103
TCP socket transmit queue depth 103
TCP socket transmit window size 103
total ARP aged entries 75
total ARP dynamic entries 75
total ARP invalid messages 75
total ARP requests received 75
total ARP requests sent 75
total ARP responses received 75
total ARP responses sent 75
total ARP static entries 75
total ICMP checksum errors 80
total ICMP ping responses received 80
total ICMP ping threads suspended 80
total ICMP ping timeouts 80
total ICMP pings received 80
total ICMP pings responsed to 80
total ICMP pings sent 80
total ICMP unhandled messages 80
total IGMP checksum errors 84
total IGMP current groups joined 84
total IGMP queries received 84
total IGMP reports sent 84
total IP bytes received 68
total IP bytes sent 68
total IP fragments received 68
total IP fragments sent 68
total IP invalid packets 68
total IP packets received 68
total IP packets sent 68
total IP receive checksum errors 68
total IP receive packets dropped 68
total IP send packets dropped 68
total packet allocations 58
total packets in pool 57
total RARP invalid messages 78
total RARP requests sent 78
total RARP responses received 78
total TCP bytes received 102
total TCP bytes sent 102
total TCP connections 102
total TCP connections dropped 102
total TCP disconnections 102
total TCP invalid packets 102

total TCP packet retransmits 102
total TCP packets received 102
total TCP packets sent 102
total TCP receive checksum errors 102
total TCP receive packets dropped 102
total UDP bytes received 90
total UDP bytes sent 90
total UDP invalid packets 90
total UDP packets received 90
total UDP packets sent 90
total UDP receive checksum Errors 90
total UDP receive packets dropped 90
UDP socket bytes received 90
UDP socket bytes sent 90
UDP socket checksum errors 90
UDP socket packets queued 90
UDP socket packets received 90
UDP socket packets sent 90
UDP socket receive packets dropped 90
status and control requests 47
status changes 47
status information 345
stop listening on a server port 100
stream data transfer between two network members 91
system configuration options 29
system equates 25
system initialization 44
system management 379
system tic division to calculate
fast TCP timer rate 36
timer rate for TCP transmit retry processing 38
timer rate for TCP-delayed ACK processing 35

T

target address space 57
target considerations 24
target hardware 24
target processor 25
target RAM 25
target ROM 24
TCP 49
TCP checksum 93
TCP checksum logic 35
TCP client connection 95
TCP client disconnection 95
TCP debug log 35
TCP disconnect protocol 97, 99
TCP enable 91
TCP for data transfer 94
TCP header 91
16-bit destination port number 91
16-bit source port number 91
16-bit TCP checksum 93
16-bit urgent pointer 93
16-bit window 93
32-bit acknowledgement number 92
32-bit sequence number 92
4-bit header length 92
6-bit code bits 93
TCP header control bits 93
TCP header format 91
TCP immediate ACK response processing
enabling 36
TCP information gathering
disabling 34
TCP keepalive timer
enabling 35
TCP output queue 53
TCP packet header 91
TCP packet queue processing 46
TCP packet receive 101
TCP packet retransmit 101
TCP packet send 100
TCP periodic processing 46
TCP ports 94
TCP queue 53
TCP receive notify 101
TCP receive packet processing 101
TCP server connection 97
TCP server disconnection 99
TCP socket control block
NX_TCP_SOCKET 103
TCP socket create 102

TCP socket state machine 94
TCP socket statistics and errors 102
TCP sockets
number of in application 102
TCP transmit queue depth before suspended or rejected TCP send request 37
TCP window size 100
thread protection 25
thread stack 25
thread stack and priority 67
thread suspension 57, 67, 80, 89, 102
ThreadX 11, 19, 21, 26, 44
ThreadX context switches 17
ThreadX mutex object 25
ThreadX RTOS 45
ThreadX support 17
ThreadX supported processors 21
ThreadX timer 25
time constraints 19
time-to-market improvement 20
total number of physical network interfaces on the host device 30
Transmission Control Protocol (TCP) 91
transmission logic 54
transmit acknowledge processing 101
transmit packet 101
transmit retries allowed before connection is broken 37
transmitting packets 348
transport layer 48
troubleshooting 27
tx_application_define 27, 44, 45
tx_port.h 11
type of ICMP message
ping request 78
ping response 80

U

UDP 49
UDP checksum 17, 86
UDP checksum calculation 45, 86
UDP checksum logic 86
UDP debug log 38
UDP enable 85
UDP Fast Path 87
UDP Fast Path Technology 17
UDP Fast Path technology 87
UDP header 85, 87
16-bit destination port number 86
16-bit source port number 86
16-bit UDP checksum 86
16-bit UDP length 86
UDP header format 85
UDP information gathering
disabling 38
UDP packet data 86
UDP packet delivery to multiple network members 81
UDP packet receive 88
UDP packet reception 87
UDP packet send 87
UDP packet transmission 85
UDP packet transmissions 88
UDP packets 48
UDP ports and binding 87
UDP receive notify 89
UDP receive packet processing 88
UDP socket 17, 87
UDP socket characteristics 90
UDP socket checksum logic 87
UDP socket control block TX_UDP_SOCKET 90
UDP socket create 89
UDP socket receive queue 17
UDP socket statistics and errors 89
UDP socket's receive queue 88
UDP utilization of IP protocol for sending and receiving packets 85
unbinding a TCP client socket from a TCP port 240
unbinding UDP socket from UDP port 338
unfragmented packets 54
unicast addresses 60
unimplemented commands 358
unique 32-bit Internet address 58
Unix host 24
upper layer protocol services 69
upper layer protocols 69

user command request 357
user commands 357
User Datagram Protocol (see UDP) 85
user-defined pointer 357
using deferred packet handling 360
using NetX 27
utilizing underlying physical media driver 58

V

version history 39

W

waiting for TCP socket to enter specific state 300
while-forever loop 46
window size 100
window size adjusted dynamically 100

Z

zero copy implementation 16
zero copy performance 55

Renesas Synergy™ Platform

User's Manual: Software

Publication Date: Rev.5 October, 2015

Renesas Synergy™ Platform

User's Manual: Software

Renesas Electronics Corporation

R11UM0004EU0500